Programmierung in C

Dr. Thomas Horn
Ingenieurhochschule Dresden,
Sektion Informationsverarbeitung

Die Programmiersprache C stellt eine moderne höhere Programmiersprache dar, die auf Grund ihrer Portabilität, Flexibilität und Effektivität, insbesondere auf 16- und 32-Bit-Rechnern, aber auch auf 8-Bit-Rechner heute eine weite Verbreitung gefunden hat. Ihr Einsatz ist nicht auf bestimmte Anwendungsgebiete begrenzt, aber durch die Möglichkeit einer maschinennahen Programmierung und hohen Laufzeiteffizienz konnte in der Systemprogrammierung die bislang dominierende Makroassemblerprogrammierung weitgehend reduziert werden. Die Systemprogrammierung ist damit ein Haupteinsatzgebiet der Programmiersprache C und begründet ihre besondere Bedeutung. Die mit diesem Beitrag eingeleitete Artikelreihe soll deshalb dem Leser eine Einführung in die Anwendung der Programmiersprache C geben.

0. Einleitung

C ist eine moderne universelle höhere Programmiersprache /1, 2/, die Anfang der siebziger Jahre von Dennis Ritchie für die Implementierung des Betriebssystems UNIX[1] auf Rechenanlagen vom Typ PDP-11 entwickelt wurde. Der für die Übersetzung der C-Programme benötigte Compiler war unter dem Betriebssystem UNIX selbst implementiert. UNIX /3/, größtenteils in C geschrieben, umfaßte etwa 13000 Quellzeilen in C und nur etwa 800 Quellzeilen in der Makroassemblersprache zur Anpassung an die Hardware, wie E/A-Driver, Interruptservice und Speicherverwaltung. Auf Grund der guten Portabilität der Programmiersprache C und der günstigen Eigenschaften von UNIX für die Softwareentwicklung ist UNIX Mitte der siebziger Jahre mit relativ wenig Aufwand auf verschiedene Groß- und Kleinrechner übertragen worden, wie IBM/360, IBM/370, Honeywell 6000, Interdata 8/32, VAX-11/780 u. a. /4, 5/. Gegenwärtig ist UNIX auf mehr als 100000 Rechenanlagen vorwiegend in der Softwareentwicklung, Forschung und Ausbildung im Einsatz.

1 Eingetragenes Warenzeichen der Bell Laboratories.

Aus der Literatur ist zu ersehen, daß UNIX mit der Entwicklung der 16-Bit-Mikroprozessoren eine starke Verbreitung als „Standard"-Betriebssystem für die 16- und 32-Bit-Mikroprozessortechnik erfahren hat /6, 7/. Infolgedessen fand auch C eine weite Verbreitung. Da die Programmiersprache aber nicht auf das Betriebssystem UNIX oder auf bestimmte Anwendungsgebiete spezialisiert ist, sondern sich durch hohe Flexibilität, Kompaktheit, Effektivität sowie moderne Konzepte für den Steuerungsfluß und für die Arbeit mit Datenstrukturen auszeichnet, ist sie insbesondere in der Systemprogrammierung als echte Alternative zur Makroassemblersprache zu betrachten. Da wie bei allen höheren Programmiersprachen die Anwendung von C nicht auf UNIX beschränkt ist, sind in den letzten Jahren zahlreiche Implementierungen von C-Compilern unter vielen Betriebssystemen auf fast allen Rechnern bekannt geworden.
Die Anpassung der Programmiersprache C an verschiedene Betriebssysteme ist vor allem deshalb relativ problemlos, da C im Vergleich zu anderen Programmiersprachen keine READ- und WRITE- bzw. GET- und PUT-Statements enthält und somit frei von Besonderheiten der E/A- und Filesteuerung ist. Der Zugriff auf die E/A- und Filesteuerung sowie auf andere Systemdienste erfolgt grundsätzlich über Funktionsaufrufe (Prozeduren), was eine außerordentlich hohe Portabilität der C-Programme bei gleichzeitig guten Laufzeiteigenschaften sichert.
Die Besonderheiten der Gerätetechnik und des Betriebssystems finden somit ihren Niederschlag nur in der Funktionsbibliothek, die auch das C-Laufzeitsystem enthält und für jeden Compiler unter den einzelnen Betriebssystemen vorhanden sein muß. Praktische Untersuchungen zeigen, daß C-Programme in der Laufzeit mit Makroassemblerprogrammen guter bis sehr guter Programmierer in etwa vergleichbar sind. Vom Speicherplatzbedarf ergeben sich bei den übersetzten C-Programmen durch das Einbinden des C-Laufzeitsystems Nachteile, die in Abhängigkeit von der Programmgröße und der Größe der Datenbereiche sowie in Abhängigkeit vom Betriebssystem im Mittel eine Vergrößerung des Arbeitsspeichers um 20 ... 50 % erforderlich machen.

1. Einführung in die Lexik

Das Alphabet der Programmiersprache C umfaßt den vollständigen Zeichensatz des ASCII-Kodes einschließlich aller Sonderzeichen und der Kleinbuchstaben (96 Zeichen). Darauf aufbauend werden in C sechs Kategorien von Begriffen unterschieden:
- *Identifikatioren*
- *Trennzeichen*
- *Schlüsselwörter*
- *Konstanten*
- *Zeichenketten und*
- *Operatoren.*

1.1. Identifikatoren

Ein *Identifikator* (Symbol) ist eine Folge von Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muß. Die ersten 8 Zeichen sind signifikant. Klein- und Großbuchstaben sind verschiedene Zeichen. Das Unterstreichungszeichen (_) zählt als Buchstabe. Externe Identifikatoren werden durch das Basis-Betriebssystem in der Länge oft eingeschränkt. Im Betriebssystem OS-RW auf Rechenanlagen des SKR sind externe Identifikatoren z. B. auf 6 Zeichen beschränkt, wobei vom Taskbuilder (Linker) auch keine Klein- und Großbuchstaben unterschieden werden.

Beispiele:
n1, block, satz_1, ALPHA, N1, _file, alpha1000, alpha1001. Die Identifikatoren **n1** und **N1** sind verschieden, während **alpha1000** und **alpha1001** identisch sind, da sie in den ersten 8 Zeichen übereinstimmen.

1.2. Trennzeichen und Kommentare

Als *Trennzeichen* werden Leerzeichen (SP), Tabulatoren (HT), Neue Zeile (NL) und Kommentare gewertet. Sie dürfen zwischen den Begriffen beliebig stehen und werden bei der Syntaxanalyse ignoriert. Falls kein anderes Sonderzeichen vorhanden ist, dienen sie nur zur Trennung der Begriffe, wie Schlüsselwörter, Identifikatoren und Konstanten.
Kommentare werden durch die Zeichenfolge /* eingeleitet und durch */ beendet. Sie dürfen an beliebiger Stelle innerhalb der Quellzeile zwischen den Begriffen an Stelle eines Trennzeichens

stehen. Eine Schachtelung von Kommentaren ist nicht zulässig. Kommentare werden im Programm zur Erläuterung der Programmanweisungen *(statements)* und somit zur Erhöhung der Lesbarkeit des Programms verwendet.

Beispiel:
int/*SPEZIFIKATION
VON I-VARIABLEN*/L,L1,L2;
L=25;/*ANFANGSWERT SETZEN*/

1.3. Schlüsselwörter

Folgende Identifikatoren sind als *Schlüsselwörter* reserviert:

asm	enum	short
auto	extern	sizeof
break	float	static
case	for	struct
char	fortran	switch
continue	goto	typedef
default	if	union
do	int	unsigned
double	long	void
else	register	while
entry	return	

Diese Schlüsselwörter dürfen anderweitig nicht noch einmal vergeben werden. Das Schlüsselwort **entry** ist gegenwärtig i.allg. noch nicht implementiert, aber für spätere Erweiterungen reserviert. In modernen C-Compilern sind auch die Schlüsselwörter **void**, **enum**, **asm** und **fortran** implementiert oder reserviert. Die Anwendung der aufgeführten Schlüsselwörter wird in den weiteren Abschnitten erläutert.

1.4. Konstanten und Zeichenketten

In C werden folgende Konstantentypen unterschieden:
- *Integerkonstanten*
- *Gleitkommakonstanten*
- *ASCII-Zeichenkonstanten und*
- *ASCII-Zeichenkettenkonstanten.*

Integerkonstanten

Integerkonstanten sind standardgemäß Dezimalzahlen. Eine führende **0** kennzeichnet Oktalzahlen, während Hexadezimalzahlen mit der Zeichenkombination **0x** bzw. **0X** beginnen. Als Hexadeziffern sind neben den Ziffernzeichen die Buchstaben **A** bis **F** bzw. **a** bis **f** vorgesehen. Die Zahl **200** kann somit als **200**, **0310** oder **0xc8** dargestellt werden.
Integerkonstanten sind standardgemäß vom Datentyp **int** ,das heißt, sie werden auf den meisten Klein- und Mikrorechnern intern im 16-Bit-Format dargestellt. Eine Dezimalkonstante, deren Wert mit Vorzeichen 16 Bit überschreitet, wird als Konstante vom Datentyp **long** (32 Bit) vereinbart. Das gleiche trifft für Oktal- und Hexadezimalzahlen zu, die ohne Vorzeichen 16 Bit überschreiten. Integerkonstanten können durch einen nachgestellten Buchstaben **1** bzw. **L** explizit im long-Format vereinbart werden.

Beispiele:
XC2FF,0177000, −1000, 0777000, 0XffcOL, +200l.

Die ersten drei Zahlen werden im 16-Bit-Format und die letzten drei Zahlen im 32-Bit-Format gespeichert.

Gleitkommakonstanten

Gleitkommakonstanten bestehen aus einem ganzzahligen Teil, dem Dezimalpunkt, einem Bruchteil, einem Buchstaben **e** oder **E** und einer Integerkonstanten (wahlweise mit Vorzeichen). Der ganzzahlige Teil oder der Bruchteil (aber nicht beide zugleich) sowie der Exponententeil (**e** bzw. **E** mit Integerkonstante) können wahlweise entfallen. Wenn ein Exponent angegeben wird, kann die Angabe des Dezimalpunktes entfallen. Gleitkommazahlen werden intern generell im Format mit höherer Genauigkeit (64 Bit) dargestellt.

Beispiele:
2E3 , 2.0 , .518 , 3.51e−10 ,−.5e+3

ASCII-Zeichenkonstanten

Eine Zeichenkonstante wird in Hochkommas, z.B. **'a'** , eingeschlossen. Ihr numerischer Wert entspricht dem Wert des Zeichens in der ASCII-Tabelle. Nichtdruckbare Zeichen, das Hochkomma (') selbst und der Backslash (\) werden mittels Escape-Folgen dargestellt, die durch den Backslash eingeleitet werden:

BS	Backspace (Rücksetzen um ein Zeichen):	\b
CR	Carriage return (Wagenrücklauf):	\r
FF	Form feed (Formularvorschub):	\f
HT	Horizontal tabulator (Tabulatorsprung):	\t
NL(LF)	Newline (Neue Zeile/ Zeilenvorschub):	\n
NUL	Null (kein Zeichen):	\0
\	Backslash (linksgeneigter Schrägstrich):	\\
'	Single quote (Hochkomma):	\'

Alle anderen nichtdruckbaren Zeichen (Steuerzeichen) werden im Oktalkode durch die Escape-Folge *ddd* vereinbart, wobei *ddd* bis zu drei Oktalziffern verkörpert. Wenn nach einem Backslash kein gültiges Zeichen folgt, so wird der Backslash ignoriert.

Beispiele: 'a', '0', 'Z', '\0', '\n', '\'', '\\', '\33', '\177'

Die Konstante '0' entspricht dem ASCII-Zeichen Null und definiert ein Byte mit dem Wert oktal 60, während '\0' ein Byte mit dem Wert 0 definiert.

ASCII-Zeichenkettenkonstanten

Eine Zeichenkettenkonstante ist eine Folge von ASCII-Zeichen, die in Anführungszeichen eingeschlossen ist, z.B. **„abcdef"**. Vom Compiler wird die Zeichenkette mit einem Null-Byte **\0** abgeschlossen (Endekennzeichen).
In Zeichenketten sind die schon beschriebenen Escape-Folgen für die Darstellung der Sonder- und Steuerzeichen zulässig (siehe Zeichenkonstanten). Ein Anführungszeichen in einer Zeichenkette muß durch die Escape-Folge \" dargestellt werden. Ein Backslash mit einem sofort folgenden Newline wird ignoriert und kann verwendet werden, um eine Zeichenkettenkonstante über mehrere Zeilen zu schreiben. Eine Zeichenkettenkonstante wird vom Datentyp „Zeichenfeld" mit der Speicherklasse **static** vereinbart, das mit den ASCII-Kodes der vorgegebenen Zeichenkette initialisiert ist. Alle Zeichenkettenkonstanten, auch wenn identisch geschrieben, werden vom Compiler getrennt gespeichert.

Beispiele:

„Das ist ein Beispiel.\n",
„Heute ist\"Montag\".\f"

1.5. Operatoren und Operanden

In C sind einstellige Operatoren, zweistellige Operatoren und Zuweisungsoperatoren vorhanden.
Einstellige Operatoren stehen in der Regel vor einem Operanden, in Ausnahmefällen auch dahinter. Zweistellige Operatoren verbinden zwei Operanden über eine arithmetische oder logische Operation. Zuweisungsoperatoren werden für die Zuweisung eines Wertes einer Variablen genutzt.
Operanden können Konstanten, Identifikatoren und aus ihnen aufgebaute Ausdrücke sein (siehe Pkt. 3.).
Ein Identifikator bezeichnet einen bestimmten Speicherbereich und wird

durch zwei Attribute charakterisiert: den Datentyp und die Speicherklasse, die im nachfolgenden Abschnitt erläutert werden. Der Speicherbereich, der an den Identifikator gekoppelt ist, wird im allgemeinen als Objekt bezeichnet. Der Datentyp legt die Bedeutung des Wertes des Objekts fest, während die Speicherklasse die Adresse und das Zeitverhalten des Objekts bestimmt.
Wie wir später sehen werden, kann man in C unter Ausnutzung des Zeigerkonzeptes (pointer) mit den Adressen von Objekten wie in der Assemblersprache arbeiten. Die Adresse eines Objekts wird über den Adreßoperator & gebildet, z.B. **&VAR** – Adresse der Variablen **VAR**. **&VAR** ist somit Ausdruck vom Zeigertyp (pointer). Der Indirektoperator * dagegen dient dem Zugriff auf die Objekte über ihre Adressen bzw. über Adreßausdrücke. Angenommen **E=&VAR**, dann muß E eine Variable vom sogenannten Zeigertyp sein, und über * E kann auf die Variable **VAR** zugegriffen werden. Es sind somit die beiden folgenden Schreibweisen identisch:

```
VAR=0
*E=0 .
```

Beide Schreibweisen zeigen, daß links vom Zuweisungsoperator (linker Teil der Ergibtanweisung) entweder einfache Variablen oder Adreßausdrücke stehen dürfen, die auf eine einfache Variable über den Indirektoperator verweisen. Wie in allen anderen Programmiersprachen dürfen rechts vom Zuweisungsoperator selbstverständlich beliebige Ausdrücke stehen.

1.6 Der Programmaufbau

Ein C-Programm ist für den Compiler eine beliebig lange Programmzeile, die aufgrund der begrenzten Zeilenlänge bei Bildschirmgeräten und Druckern beliebig in Zeilen kürzerer Länge untergliedert werden darf. C ist analog den ALGOL-ähnlichen Sprachen eine blockstrukturierte Sprache, wobei allerdings zur Erhöhung der Kompaktheit des Programms die BEGIN- und END-Klammern im Schriftbild durch geschweifte Klammern ersetzt werden. In C ist jedes Programm grundsätzlich eine Funktion, die aus dem Funktionskopf und dem Funktionsblock besteht. Funktionen sind etwa den Begriffen SUBROUTINE und FUNCTION bei FORTRAN äquivalent, unabhängig davon, ob ein Funktionswert im konkreten Fall als Ergebnis übergeben wird oder nicht.
Jedes Programm ist daher in C eine Funktion und kann als ein Baustein von anderen Funktionen aufgerufen werden. Eine Funktion, die vom Betriebssystem gestartet werden soll, muß immer den Namen **main** tragen und kann Parameter aus dem Startkommando übernehmen. Alle anderen Funktionen können beliebige Namen tragen und übernehmen ihre Parameter von der aufrufenden Funktion. Eine Funktion hat folgenden prinzipiellen Aufbau:

typ funktionsname (liste_der_formalen _parameter)
deklaration_der_formalen_parameter
{
deklarationen ;
anweisungen ;
}

wobei *typ* den Datentyp des Funktionswertes festlegt. Im Funktionsblock können lokale Variablen deklariert werden, nach deren Deklaration dann die Anweisungen folgen. In C werden alle Deklarationen und Anweisungen mit einem Semikolon (;) abgeschlossen. Ausführlich werden die Funktionen später behandelt.

Beispiel:

```
main ()
{
printf(„Hier ist unser
erstes Programm!\n");
}
```

Die Funktion **main** verfügt über keine formalen Parameter und keine lokalen Variablen. Die Funktion **printf** wird zum Ausdrucken einer Zeichenkette benutzt. Das abschließende Zeichen NL (\n) ist sehr wichtig, da es den Terminalpuffer leert und erst die Ausgabe der Zeichenkette bewirkt. Es sollte deshalb auch in den weiteren Beispielen nie vergessen werden!
Ausführlich wird die **printf**-Funktion später behandelt.

2. Einfache Datentypen und Speicherklassen

In C herrscht – ähnlich wie in ALGOL und PASCAL – ein Deklarationszwang, das heißt, alle Identifikatoren für Variablen müssen mit einem Typbezeichner und einem Speicherklassenbezeichner zu Beginn eines Blockes vereinbart werden.

2.1. Die Typbezeichner

Typbezeichner und ihre Interpretation sind:

char	für 8-Bit-Zeichen (Byte)
int oder short int	für 16-Bit-Worte mit Vorzeichen
long oder long int	für 32-Bit-Worte mit Vorzeichen
unsigned oder unsigned int	für 16-Bit-Worte ohne Vorzeichen
float	für 32-Bit-Gleitkommazahlen
double oder long float	für 64-Bit-Gleitkommazahlen

Beispiele:

```
char c1,c2;
long a,b,c;
lond float z0, z1,z2;
```

Objekte vom Datentyp **char** dienen zum Speichern von Zeichenketten. Ein Zeichen wird durch einen Integer-8-Bit-Kode (Byte) entsprechend der ASCII-Kodetabelle dargestellt.
Die Datentypen **short**, **int** und **long** bezeichnen Objekte vom Typ integer, wobei **short** und **int** dem 16-Bit-Format und **long** dem 32-Bit-Format entsprechen.
Der Datentyp **unsigned** bezeichnet vorzeichenlose Integerzahlen (16 Bit), die der modulo-65536-Arithmetik unterliegen. (Der Datentyp **unsigned long** ist nicht realisiert!) Der Datentyp **unsigned** wird insbesondere für die Speicherung von Adressen benutzt.
Die Datentypen **char** und alle Arten von **int** werden kurz als Integertypen bezeichnet.
Die Datentypen **float** und **double** realisieren Gleitkommazahlen im 32-Bit- bzw. 64-Bit-Format. Sie werden kurz als Gleitkommatypen bezeichnet.
Diese sechs genannten Datentypen sind die grundlegenden arithmetischen Typen (einfache Typen), wobei **char** als arithmetischer Typ im Byteformat betrachtet wird. Aufbauend auf die genannten einfachen Datentypen sind folgende höhere Typen möglich:

- Felder (*arrays*) von Objekten der Datentypen,
- Funktionen (**functions**), die ein Objekt mit einem vorgegebenen Datentyp zurückgeben,
- Zeiger (*pointers*) zu Objekten eines vorgegebenen Datentyps,
- Strukturen (*stuctures*), die eine Folge von Objekten verschiedener Datentypen enthalten, und
- Vereinigungen (*unions*), die verschiedenen Objekten verschiedener Datentypen den gleichen Speicherplatz zuweisen.

Die höheren Typen werden in den weiteren Abschnitten behandelt.

2.2. Definition neuer Typbezeichner

Mit dem Schlüsselwort **typedef** lassen sich neue Typbezeichner definieren. Sie können als Synonyme für andere Typ-

bezeichner oder für Strukturen bzw. Vereinigungen benutzt werden.

```
Beispiel:
typedef int tabelle, *index;
tabelle a [100], b [20];
index ai, bi;
```

Es wurden zwei neue Typbezeichner – **tabelle** und **index** – definiert, wobei **tabelle** dem Typ **int** und **index** einem Integer-Zeigertyp entsprechen. Die Felder und Zeigertypen werden erst später behandelt. Das Beispiel zeigt aber, daß **tabelle** und **index** jetzt neue Schlüsselwörter sind.

2.3. Die Speicherklassenbezeichner

Jede Variable ist in C an einen bestimmten Speicherklassenbezeichner gebunden, der die Art und Weise der Speicherplatzzuweisung für die Variablen festlegt. Vier Speicherklassenbezeichner sind vorhanden:

auto
static
extern
register

- Automatische Variablen (**auto**) sind lokale Variablen eines Blockes. Bei jedem Eintritt in einen Block wird ihnen dynamisch zur Laufzeit Speicher zugewiesen. Beim Verlassen des Blockes wird der Speicher wieder freigegeben.
- Statische Variablen (**static**) sind ebenfalls lokale Variablen eines Blokkes. Der Speicherplatz wird ihnen aber statisch zur Übersetzungszeit zugewiesen, so daß sie ihren Wert auch nach dem Austritt aus dem Block nicht verlieren. Statische Variablen, die außerhalb von Funktionen vereinbart werden, haben für alle Funktionen innerhalb eines Übersetzungsmoduls Gültigkeit.
- Externe Variablen existieren während der gesamten Programmausführung und können zum Datenaustausch zwischen den Funktionen benutzt werden. Sie müssen außerhalb von Funktionen in genau einem Übersetzungsmodul ohne Angabe eines Speicherklassenbezeichners global definiert werden. Die Bezugnahme auf global definierte Symbole erfolgt in den anderen Übersetzungsmoduln durch die Spezifikation des Speicherklassenbezeichners **extern** bei der Vereinbarung der entsprechenden Variablen. Dadurch wird ihnen kein neuer Speicherplatz zugewiesen, sondern sie werden der bereits global definierten Variablen gleichgesetzt.
- Registervariablen werden, wenn es möglich ist, in den „schnellen" Registern des Prozessors gespeichert. Registervariablen, denen kein freies Prozessorregister zugewiesen werden kann, werden wie automatische Variablen behandelt. Als Registervariablen sind nur Variablen vom Typ **int**, **unsigned**, **char** und *pointer* zulässig.

Wenn kein Speicherklassenbezeichner angegeben wird, gilt für alle Variablen innerhalb einer Funktion **auto** und außerhalb **static**.

Der Speicherklassenbezeichner wird dem Typbezeichner immer vorangestellt, zum Beispiel

static int a,b,c;

Bei dem Speicherklassenbezeichner **register** kann der Typbezeichner **int** ausgelassen werden.

2.4. Der Gültigkeitsbereich von Variablen

Variablen sind grundsätzlich immer in dem Block gültig, in dem sie definiert wurden. Bei ineinander geschachtelten Blöcken[2] sind sie auch für die inneren Blöcke gültig. Werden in einem inneren Block Variablen gleichen Namens wie im äußeren Block definiert, so gilt im inneren Block die Variable, die im inneren Block definiert wurde und im äußeren Block die im äußeren Block definierte Variable. Nur Variablen, die zu Beginn des Übersetzungsmoduls vor der ersten Funktion spezifiziert wurden, gelten für alle Funktionen des Übersetzungsmoduls ohne nochmalige Deklaration. Wurde bei solchen Variablen der Speicherklassenbezeichner **static** ausgelassen, so tragen sie zusätzlich das Attribut global. Auf globale Variablen kann von getrennt übersetzten Moduln durch Spezifikation des Speicherklassenbezeichners **extern** zugegriffen werden.

```
Beispiel:
int i;   /* Globale Variable i*/
main()
{
int j,k;   /* Automatische Variablen j
und k*/
i=10;
j=i;
incr();   /* Aufruf der Funktion incr*/
k=j+i;
printf("k=%d,j=%d  n",k,j);
}
incr()
{
extern int i;   /* Externe Variable i*/
i=i+1;
}
```

In der Funktion **printf** ist **%d** eine Formatspezifikation zum Einfügen einer Dezimalzahl. Für jedes **%d** wird som t der Wert des nächsten Parameters der Parameterliste substituiert. Als Ergebnis wird folgende Zeile auf dem Bildschirm ausgegeben:

k=21,j=10.

Literatur

/1/ Kernighan, B. W.; Ritchie, D. M.: The C Programming Language. Prentice-Hall, Inc., Englewood Cliffs, New Yersey, 1978

/2/ Kernighan, B. W.; Ritchie, D. M.: Programmieren in C. Carl-Hanser-Verlag, München, Wien, 1983

/3/ The Bell System Technical Journal. Vol. 57, Nr. 6, Part 2, (Juli – August 1978)

/4/ Lious, J.: Experience with the UNIX Time-Sharing System. Software – Practice and Experience, Vol. 9, 701–709 (1979)

/5/ Mitze, R.W.: The UNIX Systems as a Software Engineering Environments. In: Software Engineering Environments, H. Hühnke (Hrsg.), North-Holland, 1981

/6/ Lichte, I.; Harbers, H.: UNIX unterstützt Mikrocomputer-Entwicklungssystem. Elektronik, Heft 26/1982, S. 55–58

/7/ Wossidlo, M.: TNIX – eine UNIX-Implementierung für Entwicklungsaufgaben. Elektronik, Heft 26/1982, S. 61–65

/8/ Horn, Th.: Zur Anwendung der Programmiersprache C in der Systemprogrammierung. Wiss. Beiträge der IH Dresden, Heft 4, 1983, 30–42

/9/ Horn, Th.: Programmiersprachen – ein Vergleich an Hand von Beispielen. In: Kleinstrechner-TIPS, Fachbuchverlag Leipzig, 1986, 4–34

Vorschau:

Nachdem in diesem Teil der Artikelreihe zu C eine Einführung in die Lexik gegeben und die grundsätzlichen einfachen Datentypen und Speicherklassen behandelt wurden, werden im nächsten Heft die Operatoren und Ausdrücke sowie die Steuerstrukturen erläutert. Sowohl die Operatoren und Ausdrücke wie auch die Steuerstrukturen widerspiegeln maßgeblich das moderne und leistungsfähige Sprachkonzept der Programmiersprache C, das zu ihrer weiten Verbreitung in der Systemprogrammierung beigetragen hat.

2 Gemäß Sprachdefinition /1,2/ sind ineinander geschachtelte Blöcke zugelassen. Aber eine Reihe gegenwärtig weit verbreiteter Compiler unterstützt nur die Deklaration von Variablen zu Beginn des äußersten Funktionsblockes.

Programmierung in C

(Teil II)

Dr. Thomas Horn
Ingenieurhochschule Dresden
Sektion Informationsverarbeitung

Nachdem im Teil I dieser Artikelserie (s. Mikroprozessortechnik 1/87) eine Einführung in die Lexik der Programmiersprache C gegeben und die grundsätzlichen einfachen Datentypen und Speicherklassen behandelt wurden, werden nun die Operatoren und Ausdrücke sowie die Steuerstrukturen erläutert. Sowohl die Operatoren und Ausdrücke als auch die Steuerstrukturen widerspiegeln maßgeblich das moderne und leistungsfähige Sprachkonzept der Programmiersprache C, das zu ihrer weiten Verbreitung in der Systemprogrammierung beigetragen hat.

3. Operatoren, Ausdrücke und einfache Anweisungen

In C werden drei Klassen von Operatoren unterschieden:

- einstellige Operatoren,
- zweistellige Operatoren und
- Zuweisungsoperatoren.

Ausdrücke (*expressions*) werden durch Anwendung der Operatoren auf Variablen und Konstanten gebildet. Da in Ausdrücken oftmals mehrere Operatoren Anwendung finden, ist ihre Priorität und bei gleicher Priorität die Abarbeitungsreihenfolge sehr wichtig. Einstellige Operatoren haben die höchste und Zuweisungsoperatoren die niedrigste Priorität. Zur Änderung der Reihenfolge der Abarbeitung der Operatoren ist eine Klammerung mit runden Klammern zulässig.
Eine höhere Priorität als die einstelligen Operatoren haben der Funktionsaufruf *identifikator*( ), die Spezifikation von Elementen eines Feldes *identifikator*[ ] und die Auswahl von Elementen einer Struktur (− > bzw. .)
Weiterhin gibt es noch einen Kommaoperator zur Aufzählung von Ausdrükken (Aufzählungsoperator). Er liegt in der Priorität noch hinter den Zuweisungsoperatoren. Die Gesamteinordnung aller Operatoren in ein Prioritätssystem unter Angabe der Abarbeitungsreihenfolge ist in Bild 1 enthalten.

3.1. Die einstelligen Operatoren

Einstellige Operatoren stehen in der Regel vor dem Operanden und werden von rechts nach links abgearbeitet, wenn vor einem Operanden mehrere Operatoren angegeben werden. Folgende einstellige Operatoren sind definiert:

- * Indirektadressierung, z. B. ***pa** bedeutet, daß der Inhalt von **pa** als Operandenadresse zur Adressierung einer Variablen benutzt wird.
- **&** Adreßoperator, z. B. **&A [1] [2]** bedeutet Adresse des Feldelements **A [1] [2]**.
- − Zweierkomplement, z. B. **−alpha**
- ~ Einerkomplement, z. B. **~alpha**
- **!** logische Negation (Logische Variablen sind vom Typ **int** oder **char**. Der Wert 0 bedeutet *false*, und ein Wert ungleich 0 bedeutet *true*).
- ++ Inkrementoperator
- −− Dekrementoperator.

Der Inkrement- und der Dekrementoperator können vor und nach dem Operanden stehen. Vor dem Operanden geben sie ein Inkrementieren bzw. Dekrementieren vor der Speicheroperation an und nach dem Operanden das Entsprechende nach der Speicheroperation. Die Stellung der Inkrement- und Dekrementoperatoren hat damit nur bei den Feld- und Zeigeroperatoren Bedeutung; z. B. wird bei **text[i++]** auf das Zeichen **text[i]** zugegriffen, und anschließend wird der Index **i** inkrementiert, während bei **text[++i]** zugegriffen wird. Variablen vom Typ **int** und **char** werden um 1 inkrementiert bzw. dekrementiert. Zeigervariablen werden entsprechend der physischen Speicherlänge des Objekts inkrementiert bzw. dekrementiert.
Zur Realisierung einer maschinennahen, aber doch portablen Programmierung kann der einstellige Operator

sizeof (*type*) oder **sizeof** *identifikator*

verwendet werden, der die physische Speicherlänge von Datentypen oder Variablen (Felder, Strukturen usw.) ermittelt, z. B.

sizeof(int) oder **sizeof FELD**.

Ein weiterer einstelliger Operator, die explizite Typangabe **(***type***)**, kann benutzt werden, um eine bestimmte Typumwandlung zu erzwingen.

Beispiel:
sqrt((double)n)
a = (int)sqrt(7.0)

Die Funktion **sqrt** erwartet einen Operanden vom Typ **double**. Wenn **n** in einem anderen Typ vorliegt, wird durch die explizite Typangabe eine Umwandlung in das **double**-Format erzwungen. Im zweiten Beispiel wird das Ergebnis der Funktion **sqrt** in eine Integerzahl umgewandelt.

3.2. Die zweistelligen Operatoren

Zweistellige Operatoren werden nachfolgend in Gruppen nach fallender Priorität erläutert. Innerhalb der Gruppen haben die Operatoren eine gleiche Priorität und werden von links nach rechts abge-

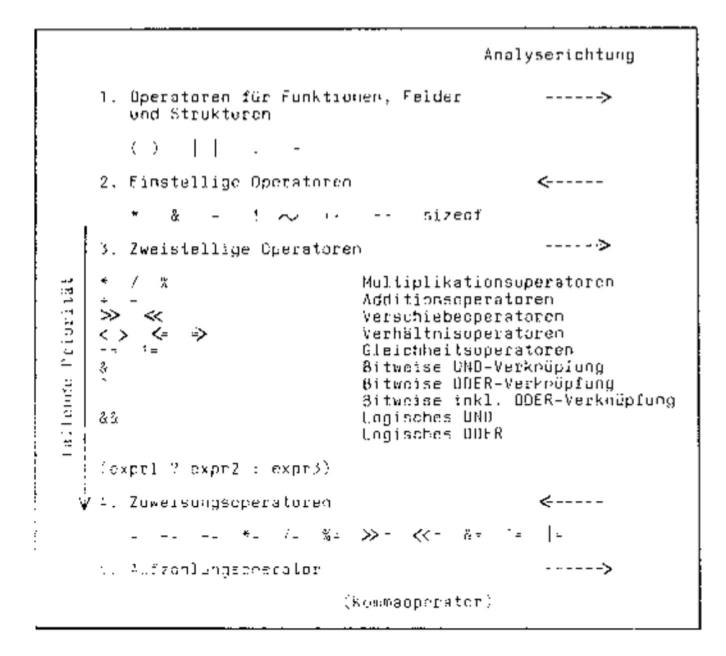


Bild 1 Prioritäten und Analyserichtung der Operatoren

arbeitet, wenn in einem Ausdruck mehrere zweistellige Operatoren angegeben sind.

- **Multiplikationsoperatoren:**
  - * Multiplikation
  - / Division
  - % Rest von der ganzzahligen Division
- **Additionsoperatoren:**
  - \+ Addition
  - − Subtraktion
- **Verschiebeoperatoren:**
  - >> Rechtsverschiebung, z. B. **alpha>>2** (2 Bit nach rechts)
  - << Linksverschiebung, z. B. **alpha<<A0** (A0 Bit nach links)
- **Verhältnisoperatoren:**
  - < kleiner als
  - > größer als
  - <= kleiner oder gleich
  - >= größer oder gleich
- **Gleichheitsoperatoren:**
  - == gleich
  - != ungleich
- **Bitweise UND-Verknüpfung:**
  - & Die einzelnen Bits der Variablen vom Integertyp werden bitweise konjunktiv (logisches UND) verknüpft.
- **Bitweise exklusive ODER-Verknüpfung:**
  - ∧ Die einzelnen Bits der Variablen vom Integertyp werden bitweise entsprechend der Funktion Antivalenz (exklusives logisches ODER) verknüpft.
- **Bitweise inklusive ODER-Verknüpfung:**
  - | Die einzelnen Bits der Variablen vom Integertyp werden bitweise disjunktiv (inklusives ODER) verknüpft.
- **Logisches UND:**
  - && Die Werte der Variablen werden als logische Werte *true* oder *false* interpretiert und konjunktiv (logisches UND) verknüpft.
- **Logisches ODER:**
  - || Die Werte der Variablen werden als logische Werte *true* oder *false* interpretiert und disjunktiv (logisches ODER) verknüpft.
- **Bedingter Operator:**
  expression1 **?** *expression2* **:** *expression3*

Wenn der Ausdruck *expression1 true* (ungleich Null) ist, so gilt *expression2*, andernfalls *expression3*.

Beispiel:
V= a0 < a1 ? a0 : a1;

Der Variablen **V** wird entweder **a0** oder **a1** zugewiesen, in Abhängigkeit davon, welche Variable den kleineren Wert hat.

3.3. Die Zuweisungsoperatoren

Der generelle Zuweisungsoperator ist wie in anderen Programmiersprachen das Gleichheitszeichen (=). Da sehr häufig zweistellige Operationen mit einer Zielvariablen vorkommen, werden zur Erhöhung der Prägnanz weitere Zuweisungsoperatoren zugelassen:

+= −= *= /= %= <<=
>>= &= ∧= |=

Auf der linken Seite der Zuweisungsoperatoren muß immer ein Ausdruck stehen, dem ein Wert zugewiesen werden kann, also eine einfache Variable oder ein Adreßausdruck mit dem Indirektoperator (vergleiche auch 1.5.).

Beispiele:
a+=2; /* entspricht a=a+2 */
b<<=5; /* entspricht b=b>>5 */

Außerdem werden in einer Anweisung mehrere Zuweisungsoperatoren zugelassen, die dann von rechts nach links abgearbeitet werden, zum Beispiel

a+=b*=c=d>>=2;

bedeutet:

a=a+b*(d>>2);
b=b*(d>>2);
c=d>>2;
d=d>>2;

3.4. Anweisungen

Ausdrücke werden zur Anweisung (*statement*), wenn sie durch ein Semikolon abgeschlossen werden, zum Beispiel

a=b=0;
j++;
putc(n);
; /* Leeranweisung */

In C wird eine Anweisung durch ein Semikolon beendet, im Gegensatz zu PASCAL, wo ein Semikolon zwei Anweisungen trennt. Durch die Verwendung von geschweiften Klammern können mehrere Anweisungen zu einer Verbundanweisung (*compound statement*) zusammengefaßt werden, die – syntaktisch gesehen – wie eine Anweisung behandelt wird. Nach der rechten geschweiften Klammer steht niemals ein Semikolon. Der Unterschied zwischen einer Verbundanweisung und einem Block besteht darin, daß in einer Verbundanweisung keine neuen Variablen deklariert werden.

4. Steuerstrukturen

Steuerstrukturen werden innerhalb von Funktionen verwendet, um die Reihenfolge der Abarbeitung von Anweisungen festzulegen. In den einzelnen Steuerstrukturen darf in der Regel eine Anweisung (*statement*) stehen. Sollen mehrere Anweisungen abgearbeitet werden, so sind diese durch geschweifte Klammern zu einer Verbundanweisung (*compound statement*) zusammenzufassen. Nach der linken geschweiften Klammer dürfen Deklarationen für neue Variablen stehen, so daß anstelle einer einzelnen Anweisung tatsächlich auch immer ein Block stehen darf.[3]
Unter Verwendung der weitverbreiteten Struktogrammtechnik zur grafischen Darstellung der Programmstrukturen kann eine Funktion im allgemeinen als eine Sequenz von *Strukturblöcken* (Bild 2) dargestellt werden. Im Sinne von C ist jeder *Strukturblock* eine einzelne Anweisung, eine Verbundanweisung oder ein Block.

---
[3] siehe Fußnote 2 in Teil I

Strukturblock 1
Strukturblock 2
...
Strukturblock n

Bild 2
Sequenz von Strukturblöcken

if(cond)
true false
Strukturblock Strukturblock

Bild 3
Vollständige Alternative

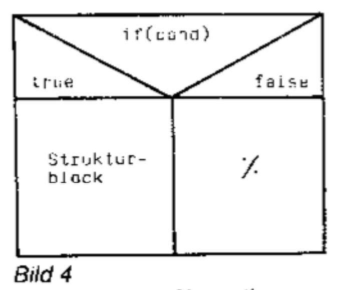


Bild 4
Unvollständige Alternative

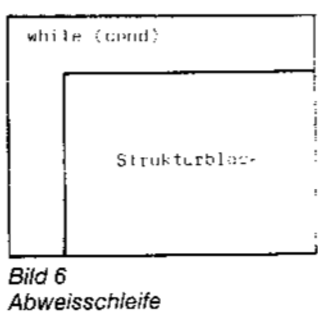


Bild 6
Abweisschleife

4.1. Die bedingte Anweisung

Die bedingte Anweisung (Alternative) ist als **if-else**-Anweisung realisiert, wobei der **else**-Teil wahlfrei ist. Bild 3 zeigt das Struktogramm für die vollständige und Bild 4 für die unvollständige Alternative. Die Syntax der bedingten Anweisung ist:

if (*expression***)** *statement1* **else** *statement2*

Der Ausdruck *expression* wird berechnet und als Verzweigungsbedingung *cond* bewertet. Ist das Ergebnis ungleich 0 (*true*), so wird *statement1* (linker Strukturblock) ausgeführt. Wenn das Ergebnis gleich 0 (*false*) ist, so wird, falls vorhanden, die Anweisung *statement2* im **else**-Teil (rechter Strukturblock) ausgeführt.

Beispiel:
if (n−1) f=f∗(n−1); else f=1;
oder
if (n==1) f=1; else f=f∗ (n−1);

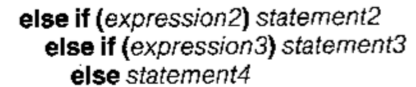
Sowohl im **if**-Teil als auch im **else**-Teil dürfen im auszuführenden Strukturblock weitere **if**-Anweisungen stehen. Da der **else**-Teil wahlfrei ist, besteht bei geschachtelten **if**-Anweisungen die Gefahr, daß die **else**-Teile in der Programmnotation nicht eindeutig zugeordnet werden können. Vom Compiler wird deshalb ein **else**-Teil immer der letzten **if**-Anweisung zugeordnet, die noch keinen **else**-Teil hat.

Beispiel:
if (n<0) if (x<y) z=x;
else z=y;

Der **else**-Teil entspricht der zweiten **if**-Anweisung. Eine Zuordnung zur ersten **if**-Anweisung kann durch geschweifte Klammern erzwungen werden:

if (n<100) {if (x<y) z=x;}
else z=y;

Wesentlich günstiger ist die Anordnung weiterer **if**-Anweisungen im **else**-Teil, z. B.:
if (*expression1***)** *statement1*


else if (*expression2***)** *statement2*
else if (*expression3***)** *statement3*
else *statement4*

Diese Folge von **if**-Anweisungen ist die allgemeingültige Möglichkeit, unter vielen Alternativen eine Möglichkeit herauszusuchen. In der angegebenen Reihenfolge werden die Ausdrücke berechnet, und falls ein Ausdruck *true* ist, wird die entsprechende Anweisung ausgeführt. Der letzte **else**-Teil behandelt den Fall, daß keine der vorherigen Bedingungen zutraf. Er kann dazu benutzt werden, den ‚allgemeinen' Fall oder eine ‚unmögliche' Bedingung zu realisieren. Falls nicht erforderlich, kann der letzte **else**-Teil auch entfallen.

4.2. Die Fallauswahl

Die Fallauswahl (Bild 5) ist eine Erweiterung der Alternative auf mehr als zwei Fälle. Sie wird über die **switch**- und **case**-Anweisungen realisiert. In Abhängigkeit vom Wert der Bedingung *cond* (Schalterausdruck *expression*) kann einer von mehreren folgenden Strukturblöcken ausgewählt werden. Die **switch**-Anweisung hat folgende Form:

switch (*expression***)** *statement*

Das Resultat des Schalterausdrucks muß ein Integer-Wert sein. Die abhängige Anweisung *statement* ist zweckmäßigerweise eine Verbundanweisung. Jeder Anweisung innerhalb der Verbundanweisung kann eine beliebige Anzahl von **case**-Marken vorausgehen:

case *constant-expression***:**

Nach dem Schlüsselwort **case** muß ein Integer-Wert stehen, der durch einen Konstantenausdruck vorgegeben werden kann; das heißt, der Ausdruck muß zur Compilierzeit berechenbar sein. Jeder Konstantenausdruck muß einen anderen Integer-Wert ergeben.
Für alle übrigen, nicht spezifizierten Fälle wird die **case**-Marke

default:

verwendet.
Wenn für einen Schalterwert keine entsprechende **case**-Marke vorhanden und auch keine **default**-Marke spezifiziert ist, so wird die abhängige Anweisung nicht ausgeführt und die nächste Anweisung nach der **switch**-Anweisung abgearbeitet.
Die **case**-Marken selbst haben keinen Einfluß auf die sequentielle Abarbeitung der Anweisungen in der Verbundanweisung. Sie stellen lediglich verschiedene Einsprungmarken dar. Zum Verlassen eines **case**-Teils wird normalerweise die **break**-Anweisung benutzt. Die **case**-Marke spezifiziert somit den Anfang und die **break**-Anweisung das Ende eines Strukturblockes.

Hinweis:

Da über die **case**-Marken und nicht über die geschweifte Klammer der Eintritt in die Verbundanweisung der **switch**-Anweisung erfolgt, werden keine Laufzeitinitialisierungen von Variablen ausgeführt. Aus diesem Grunde sollten in der Verbundanweisung keine Variablen über die Speicherklasse **auto** oder **register** initialisiert werden.

Beispiel:
Zählen der Leerzeichen, Punkte, Kommas und Zeilenvorschübe in einem Text.

```
main()
{
static int nl=0, np=0, nk=0,
nz=0, nr=0; char c;
while ((c=getchar())!=EOF)
switch(c) {
case ' ': nl++;break;
case '.': np++;break;
case ',': nk++;break;
case '\n': nz++;break;
default:  nr++;break;
}
printf("\nL=%d, P=%d, K=%d,
Z=%d, Rest=%d\n", nl, np,
nk, nz, nr);
}
```

Die Funktion **getchar** stellt immer das nächste Zeichen aus dem Terminalpuffer zur Verfügung. Die Eingabe in den Terminalpuffer erfolgt mittels Drücken der (alphanumerischen) Tasten des Terminals und wird mit der RETURN-Taste abgeschlossen. Das Zeichen **EOF** (End of File) wird durch das Steuerzeichen

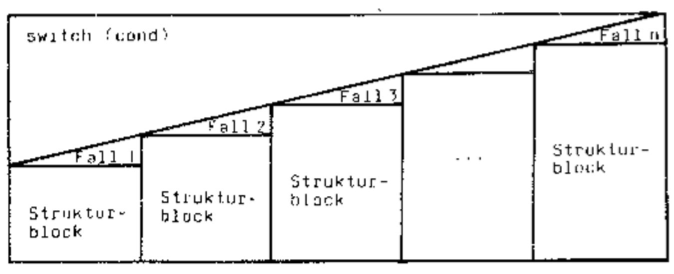


Bild 5 Fallauswahl

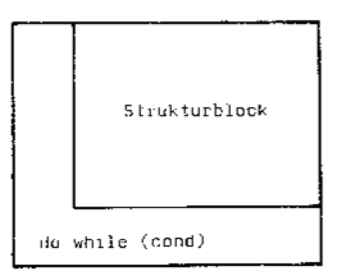


Bild 7 Nichtabweisschleife

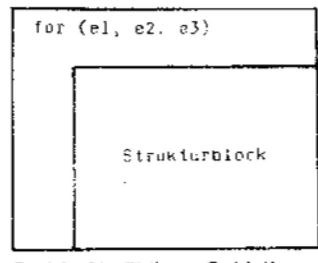


Bild 8 Abzählbare Schleife

CTRL/Z und RETURN eingegeben. Die **while**-Anweisung wiederholt die **switch**-Anweisung bis zum **EOF**.

4.3. Die Abweisschleife

Die Abweisschleife (Bild 6) wird über die **while**-Anweisung realisiert. Sie hat folgende Syntax:

while (*expression***)** *statement*

Der Ausdruck *expression* wird berechnet. Ist sein Wert ungleich 0 (*true*), so wird die abhängige Anweisung *statement* ausgeführt und die **while**-Anweisung wiederholt. Ist der Wert des Ausdrucks 0 (*false*), so wird die Schleife beendet. Bei der Wiederholung der **while**-Anweisung wird der Ausdruck *expression* neu berechnet. Ist die Anweisung eine Verbundanweisung, so kann mittels **continue**-Anweisung vorzeitig zur Wiederholung der **while**-Anweisung übergegangen werden. Durch eine **break**-Anweisung kann die **while**-Anweisung vorzeitig verlassen werden.

Beispiel:
Berechnen der Fakultät **f** einer Zahl **i**.

```
f=1; i=6;
while (i) {f=f*i;--i;)}
oder
while (1) {if(i<=1)break;
f*=i;i--;}
```

Im zweiten Beispiel würde die **while**-Anweisung durch die Bedingung **(1)** unendlich oft wiederholt werden. Zum Verlassen der Schleife wird die **break**-Anweisung benutzt.

4.4. Die Nichtabweisschleife

Die Nichtabweisschleife (Bild 7) wird als **do-while**-Anweisung realisiert. Sie hat folgende Syntax:

do *statement* **while (***expression***);**

Zuerst wird die Anweisung *statement* ausgeführt und anschließend der Ausdruck *expression* berechnet. Ist der Ausdruck ungleich 0 (*true*), so wird die Schleife wiederholt. Wenn der Ausdruck gleich 0 (*false*) ist, so wird die Schleife beendet. Ist die Anweisung eine Verbundanweisung, so kann mit einer **continue**-Anweisung vorzeitig zur Wiederholung der **do**-Schleife übergegangen werden, während mittels **break**-Anweisung die Schleife vorzeitig verlassen wird.

Beispiel:
Berechnen der Fakultät **f** einer Zahl **i**.

```
f=1; i=6; do f=f*i;
while (--i==0);
```

4.5. Die verallgemeinerte abzählbare Schleife

Die verallgemeinerte abzählbare Schleife (Bild 8) wird durch die **for**-Anweisung realisiert. Sie hat folgende Syntax:

for (*expression1***;***expression2***;***expression3***)** *statement*

Die **for**-Anweisung kann als eine äquivalente **while**-Schleife wie folgt dargestellt werden:

*expression1***;while (***expression2***)**
{*statement expression3***;**}

Der Ausdruck *expression1* dient zur Initialisierung der Schleife. Der Ausdruck *expression2* wird vor jeder Ausführung der abhängigen Anweisung *statement* berechnet und bewirkt bei ungleich 0 (*true*) die Ausführung der abhängigen Anweisung. Bei gleich 0 (*false*) wird die Schleife beendet. Der Ausdruck *expression3* wird typischerweise zur Berechnung des nächsten Wertes der Laufvariablen verwendet.
Jeder Ausdruck kann in der Anweisung ausgelassen werden. Bei ausgelassenem Ausdruck *expression2* gilt **while (1)**. Die Schleife muß dann mit einer **break**-, **goto**- oder **return**-Anweisung verlassen werden. Fehlen die Ausdrücke *expression1* und *expression3*, so gelten sie als nicht spezifiziert.

Beispiel:
Suchen des ersten Kommas im Textpuffer **text [100]**:

```
for(i=0;text[i]!='\0';i++)
if(text[i]==',')break;
oder
for(i=-1;text[++i]!='\0';)
if(text[i]==',')break;
oder
for(i=0;text[i]!='\0'
&(text[i]!=',';i++);
```

Der Textpuffer **text[ ]** ist ein Feld. Der Index **i** wird ab Null beginnend gezählt. Mit **text [i]** wird auf das **i**-te Zeichen des Textpuffers zugegriffen. Der Ausdruck **text[++i]** gibt an, daß vor dem Zugriff auf das **i**-te Zeichen der Wert von **i** um Eins erhöht wird. Das Zeichen **\0** gibt das Ende der Zeichenkette an.

4.6. Die Fortsetzungsanweisung

Die Anweisung
continue;
kann sich in den abhängigen Anweisungen einer **while**-, **do**- oder **for**-Anweisung befinden und bewirkt den vorzeitigen Übergang zur nächsten Schleifenausführung, das heißt, es wird mit dem Test auf die Schleifenwiederholung fortgesetzt. Bei der **for**-Anweisung wird aber vorher noch der Ausdruck *expression3* berechnet.

4.7. Die Unterbrechungsanweisung

Die Anweisung

break;

kann sich in den abhängigen Anweisungen einer **while**-, **do**- oder **for**-Anweisung befinden und bewirkt das vorzeitige bedingungslose Verlassen der Schleife, das heißt, es wird mit der nächsten Anweisung, die nach der Schleife folgt, die Programmabarbeitung fortgesetzt.

4.8. Die Sprunganweisung

Die Anweisung

goto *label***;**

bewirkt einen Sprung zu einer Marke *label*. Die Marke muß sich in einem Block der gleichen Funktion befinden. Marken sind *Identifikatoren*, die vor einer beliebigen Anweisung definiert sind:

*label***:**

Vor einer Anweisung kann eine beliebige Anzahl Marken stehen. Der Geltungsbereich der Marken erstreckt sich über die ganze Funktion. Eine Ausnahme sind verschachtelte Blöcke einer Funktion, in denen der gleiche Identifikator neu als Marke definiert wird. Im allgemeinen kann ein Programm ohne die **goto**-Anweisung geschrieben werden. Da die Anwendung der **goto**-Anweisung den Prinzipien der strukturierten Programmierung widerspricht, sollte man auf ihre Anwendung im Interesse einer klaren Programmstruktur verzichten.

Beispiel:
In diesem Beispiel soll gezeigt werden, wie die **goto**-Anweisung sinnvoll in der Systemprogrammierung eingesetzt werden kann, um bei Fehlerzuständen mehrere ineinander geschachtelte Strukturblöcke effektiv zu verlassen.

```
for(...)
{
while(...)
  switch(...) {
  case...:
  case...:...
          if(rc<0) goto ERROR;
          ...
  default:...
  }
}
...
ERROR: /*Fehleranalyse-
routine*/
...
```

Programmierung in C

(Teil III)

Dr. Thomas Horn
Informatik-Zentrum des Hochschulwesens an der Technischen Universität Dresden

Nachdem in den ersten beiden Teilen dieser Artikelreihe die Lexik, Datentypen, Speicherklassen, Operatoren, Ausdrücke und Steuerstrukturen erläutert wurden, werden in diesem Teil das Zeigerkonzept und die Arbeit mit Feldern beschrieben. Insbesondere das Zeigerkonzept ist für die Systemprogrammierung von Bedeutung, da erst die Arbeit mit Adressen eine effektive Lösung vieler Probleme gestattet und auch gegenüber der Arbeit mit Feldindizes Vorteile hat. Hierin drücken sich vor allem die Vorteile von C als effektive Systemprogrammiersprache aus.
Die Felder dienen zur Zusammenfassung von Datenobjekten gleichen Typs. Im weiteren werden dann die Strukturen und Vereinigungen zur Zusammenfassung von Datenobjekten verschiedener Typen behandelt.

5. Zeiger

Die Programmiersprache C enthält ein ausgezeichnetes Zeigerkonzept. Unter einem Zeiger (*pointer*) versteht man ein Datenobjekt, das eine Adresse enthält, über die auf beliebige andere Datenobjekte zugegriffen werden kann. Intern werden Zeiger wie **unsigned**-Variablen dargestellt. Sie dürfen aber nicht mit **unsigned**-Variablen verwechselt werden, da Zeiger zusätzlich über ein Längenattribut verfügen, das die Länge des Datenobjekts in Byte spezifiziert, auf das der Zeiger zeigt. Das Längenattribut ist insbesondere für Adreßrechnungen wichtig. Das Zeigerkonzept wird vor allem in der Systemprogrammierung genutzt, weil viele Probleme nur über die Verwendung von Adressen effektiv gelöst werden können. Auch in anderen Anwendungen bringt die Arbeit mit Adressen Effektivitätsvorteile gegenüber der Arbeit mit Feldindizes.
Das Zeigerkonzept birgt jedoch generell die Gefahr in sich, daß durch die Arbeit mit Adressen die Portabilität der Programme eingeschränkt wird. In C ist das Zeigerkonzept so implementiert, daß bei einer sorgfältigen Programmierung eine Verletzung des Portabilitätsprinzips ausgeschlossen werden kann.

5.1. Zeigervariablen

Wie alle Variablen müssen Zeigervariablen zu Beginn eines Blockes vereinbart werden, wobei der Indirektoperator (*) vor dem Identifikator die Variable als Zeigervariable und der Typbezeichner ihren Typ spezifizieren.

Beispiel:
```
int * pi;
char * pc;
float * pf;
```

Die Variable **pi** ist ein Zeiger auf eine Integervariable, **pc** ein Zeiger auf eine Zeichenvariable und **pf** ein Zeiger auf eine Gleitkommazahl einfacher Genauigkeit.
Adressen werden von einfachen Variablen über den Adreßoperator (**&**) gebildet.

Beispiele:
```
pi=&A;
pc=&buffer [0];
pf=&sum;
```

Der Zugriff auf die Speicherobjekte erfolgt dann über einen Zeiger durch Anwendung des Indirektoperators (*) auf die Zeigervariable.

Beispiel:
```
pi=&A;
B=*pi;
```

Durch die letzte Anweisung wird der Variablen **B** der gleiche Wert wie bei der Anweisung **B=A** zugewiesen. Da **pi** als ein Integerzeiger vereinbart ist, besagt die Kombination ***pi**, daß das Speicherobjekt ein Integerwert ist.

Zeigervariablen können auch in Ausdrücken auftreten.

Beispiele:
```
B=*pi+1;
```

Durch diese Anweisung wird das Speicherobjekt ***pi** gelesen und anschließend um 1 erhöht. Das Ergebnis entspricht somit **B=A+1**

```
B=*(pi+1);
```

Durch diese Anweisung wird der Zeiger **pi** um 1 erhöht. Da es sich um einen Integerzeiger handelt, wird der Zeiger auf das nächste Integerobjekt weiter gestellt (Erhöhung der Adresse um 2 Byte), welches dann gelesen wird.
Diese Operation wird oft bei Feldern angewendet, um das nächste Feldelement zu adressieren.
Zeiger können auch auf der linken Seite von Zuweisungen verwendet werden.

Beispiel:
```
*pi *=5;
```

Das Speicherobjekt vom Typ **int**, das durch **pi** adressiert wird, wird mit fünf multipliziert und auf den selben Speicherplatz abgelegt.

Hinweis:

Das Zeigerkonzept kann nicht auf Variablen der Speicherklasse **register** angewendet werden, da von Registern keine Adressen gebildet werden können.

5.2. Adreßrechnungen

Unter Adreßrechnungen werden Operationen mit den Adressen verstanden, die in den Zeigervariablen gespeichert sind. Bei den Adreßrechnungen ist zu beachten, daß alle arithmetischen Operationen unter Beachtung der Längenattribute der Zeiger ausgeführt werden.

Die Adreßarithmetik läßt folgende Operationen mit Zeigern zu:

- Addition einer ganzen Zahl zum Zeigerwert
  Der Zeiger wird um eine ganze Zahl von Datenobjekten vorwärts gerückt, z. B. **pf+=5;** oder **pi++;**
- Subtraktion einer ganzen Zahl vom Zeigerwert
  Der Zeiger wird um eine ganze Zahl von Datenobjekten rückwärts gesetzt, z. B. **pc−=3;** oder **pf−−;**
- Subtraktion von zwei Zeigern
  Die Differenz von zwei Zeigern ist die Anzahl von Datenobjekten, die sich zwischen zwei Zeigern befinden, z. B. **i=pf2−pf1;**
- Vergleich von zwei Zeigern
  Der Vergleich von zwei Zeigern ergibt, ob zwei Zeiger identisch sind oder nicht bzw. ob ein Zeiger größer oder kleiner als der andere ist, z. B. **if (pf1 < pf2)...;**

Hinweise:

– Zwei Zeiger können nicht addiert werden.
– Zeiger können nicht multipliziert, dividiert, verschoben oder mit anderen logischen Operationen verändert werden.
– Es können keine Additionen oder Subtraktionen mit **float-** oder **double-**Werten ausgeführt werden.

Beispiel:
Es seien die Zeigervariablen **pi**, **pc** und **pf** definiert (siehe 5.1.).

```
pi++;
pc++;
pf++;
```

Der Inkrementoperator **++** führt eine Erhöhung der Zeiger auf das nächste Speicherobjekt durch. Bei **pi** wird somit die Adresse um zwei Bytes, bei **pc** um ein Byte und bei **pf** um vier Bytes erhöht. (Die Länge eines Speicherobjektes kann mit dem **sizeof**-Operator bestimmt werden.)
Zur Absicherung der richtigen Ausführung von Adreßoperationen kann von der expliziten Zeigerumwandlung Gebrauch gemacht werden.

Beispiel:

```
pc=(char*)pi;
```

Der Zeiger vom Typ **int** wird in einen Zeiger vom Typ **char** umgewandelt.

Analog können auch Integerwerte in Zeiger und umgekehrt umgewandelt werden. Auf verschiedenen Rechenanlagen können hierbei Wortgrenzenfehler entstehen. Deshalb sollten die Zeigerumwandlungen mit größter Vorsicht verwendet werden.

5.3. Initialisierung von Zeigern

Im allgemeinen kann ein Zeiger wie jede andere Variable initialisiert werden. Sinnvolle Werte sind allerdings nur die Adressen bereits vorher definierter Objekte. In C ist sichergestellt, daß nicht initialisierte Zeiger bzw. Zeiger, die nicht korrekt auf ein Speicherobjekt verweisen, den Wert 0 erhalten. Die Adresse 0 ist niemals die Adresse eines gültigen Speicherobjekts. Man kann deshalb auch den Wert 0 benutzen, um eine Fehlersituation anzuzeigen. Damit kann über den Zeiger auf kein Speicherobjekt mehr zugegriffen werden. Um sichtbar zu machen, daß der Wert 0 eine besondere Bedeutung hat, wird das Symbol **NULL** verwendet,

$A_{0,0}$	2
$A_{0,1}$	3
$A_{0,2}$	5
$A_{1,0}$	1
$A_{1,1}$	4
$A_{1,2}$	6

Bild 1 Anordnung der Feldelemente im Speicher

das im File **STDIO.H** (siehe 10.1.) definiert ist.

Beispiel:

```
char text [2000];
static char *t=&text [0];
```

Durch die Initialisierung wird der Zeiger **t** auf den Anfang des Textpuffers **text** gesetzt.

```
if (t==&text [1999]) t=NULL;
```

Wenn das Ende des Textpuffers erreicht ist, wird **t** auf **NULL** gesetzt, was an späterer Stelle im Programm wieder getestet werden kann:

```
if (t==NULL) goto ENDE;
```

6. Felder

Felder werden zur Zusammenfassung von Datenobjekten des gleichen Typs zu einer Gesamtheit verwendet. Die Datenobjekte werden dann unter dem gemeinsamen Feldnamen geführt. Auf die einzelnen Datenobjekte, auch Feldelemente genannt, kann über Indizes zugegriffen werden. Werden die Feldelemente linear angeordnet, so können sie über einen Index adressiert werden. Man spricht von einem eindimensionalen Feld. Bei einer matrizenähnlichen Anordnung, einem zweidimensionalen Feld, sind für den Zugriff zwei Indizes erforderlich. Analog sind auch mehrdimensionale Felder mit mehreren Indizes möglich.

6.1. Ein- und mehrdimensionale Felder

In C werden (eindimensionale) Felder durch Angabe der Dimension hinter dem Identifikator, der im weiteren ein Feld bezeichnen soll, deklariert. Die Dimension sowie der Index zum Zugriff auf die Feldelemente werden in eckigen Klammern angegeben. Der Index wird – wie in der Systemprogrammierung – ab Null beginnend gezählt.

Beispiel:

```
char text [4];
text [0] "'a';
text [1] ='b';
text [2] ='c';
text [3] ='\0';
```

Es wird das Feld **text**, bestehend aus vier Elementen vom Typ **char**, vereinbart. Den vier Feldelementen werden anschließend ASCII-Zeichen zugewiesen.
Bei mehrdimensionalen Feldern werden die Dimensionen und die Indizes jeweils getrennt in eckigen Klammern angegeben. Die Anordnung der Feldelemente im Speicher erfolgt zeilenweise, das heißt, beim Übergang von einem Element zum nächsten ändert sich der letzte Index am schnellsten (Umgekehrt als in FORTRAN!).

Beispiel:

```
static int A [2] [3] =
{{2,3,5}, {1,4,6}};
```

Es wird ein zweidimensionales Feld vom Typ **int** vereinbart, wobei der erste Index die Werte 0 und 1 und der zweite Index die Werte 0, 1 und 2 annehmen kann. Den einzelnen Elementen werden Anfangswerte zugewiesen. Die Anordnung der Feldelemente mit ihren Anfangswerten ist in Bild 1 dargestellt.
Die Summe aller Feldelemente kann durch folgende Anweisung berechnet werden:

```
for (s=i=0;i<2;i++)
for (j=0;j<3;j++) s=A [i] [j];
```

Hinweis:

In C sind nur Operationen für die einfachen Datentypen definiert, das heißt, es gibt keine Operationen für ein ganzes Feld, sondern nur für die Feldelemente, sofern die Feldelemente einfache Datenobjekte sind.

6.2. Zeiger auf Felder

Zeiger werden oft zur Verarbeitung von Feldern benutzt. Zeiger können auf den Feldanfang und auf einzelne Feldelemente gesetzt werden. In C ist der Feldname identisch mit der Adresse des Feldes und entspricht somit der Adresse des nullten Feldelementes. Die Adresse muß über den Adreßoperator (**&**) gebildet werden.

Beispiel:

```
int A [10], *a,i;
a=A;
```

Der Zeiger **a** wird auf das Feld **A** gesetzt. Da **A [0]** das nullte Feldelement ist, führt die folgende Anweisung

a=&A [0];

zum gleichen Resultat. Mit der Anweisung

a=&A [i];

wird der Zeiger **a** auf die Adresse des i-ten Feldelementes gesetzt.

Hinweis:

Es besteht ein sehr enger Zusammenhang zwischen der Indizierung und der Zeigerarithmetik. Der Compiler verwandelt einen Verweis auf ein Feld in einen Zeigerwert. Dadurch ist ein Feldname immer ein Zeigerausdruck (siehe **a=A;**), und der Adreßoperator (**&**) ist in diesem Sonderfall nicht erforderlich.

Der Zugriff auf die einzelnen Feldelemente erfolgt über den Indirektoperator (*). Da es keine Operationen gibt, die sich auf Felder im ganzen beziehen, kann der Feldname als Zeiger auf den Feldanfang benutzt werden.

Beispiel:
a=A;
b=*a; oder **b=*A;**
c=*(a+i); oder **c=*(A+i);**

Der Variablen **b** wird der Wert des Elements **A [0]** und der Variablen **c** der Wert des Elements **A [i]** zugewiesen. Die Schreibweise ***(A+i)** ist somit ein Äquivalent für die indizierte Schreibweise **A [i]**.

6.3. Zeigerfelder

Mehrere Zeiger können zu Zeigerfeldern zusammengefaßt werden.

Beispiel:
int *a [4], A [10];

Das Feld **a** ist ein Zeigerfeld vom Typ **int**, bestehend aus vier Zeigern.

a [0] =&A [0];
a [1] =&A [2];
a [2] =&A [5];
a [3] =&A [9];

Den vier Zeigern werden die Adressen der Elemente **A[0], A[2], A[5]** und **A[9]** zugewiesen.

Die Zeiger des Zeigerfeldes können durch Angabe des Index verwendet werden.

Beispiel:
b=*a[0] + *a[2] − *a[3];
Der Ausdruck ist dem Wert von **A [0] + A [5] − A [9]** äquivalent.

Hinweis:

Entsprechend dem rekursiven Grundprinzip der Programmiersprache C können Zeiger auf Zeigerfelder gesetzt werden , die wiederum zu Zeigerfeldern zusammengefaßt werden können.

6.4. Zeichenfelder und Zeichenketten

Auch Zeichen können in Feldern angeordnet werden. Jedes Element eines Zeichenfeldes ist dann ein ASCII-Zeichen. In C gibt es keinen Zeichenkettentyp. Deshalb werden Zeichenketten als Zeichenfelder gespeichert. Trotzdem gibt es einen wichtigen Unterschied zwischen Zeichenketten und Zeichenfeldern, denn Zeichenketten müssen in C immer mit einem Nullzeichen **\0** enden. Bei der Definition von Zeichenkettenkonstanten wird vom Compiler das Nullzeichen **\0** automatisch angefügt. Somit sind **'1'** und **"1"** in C verschiedene Definitionen! Die Länge der Zeichenkette im Speicher ist daher immer um ein Zeichen größer. Auf Zeichenkettendefinitionen wird wie bei Feldern über Zeiger zugegriffen. Der Zeiger auf eine Zeichenkette verweist immer auf das erste Zeichen der Zeichenkette.

Beispiel:
char * text1;
text1 = "Ende des Programms TEXT1\n";

Vom Compiler wird die Zeichenkette als Konstante im Hauptspeicher abgelegt. Die Adresse der Zeichenkette wird dem Zeiger **text1** zugewiesen. Die Zeichenkette wird also nicht kopiert, denn an der Zuweisungsoperation sind nur Zeiger beteiligt.

Hinweis:

Die Programmiersprache C hat keine Operatoren, die eine Zeichenkette als ein Objekt behandeln, da es nur Operationen für die einfachen Datentypen gibt.

6.5. Initialisierung von einfachen Variablen und Feldern

Einfache Variablen können bei ihrer Definition grundsätzlich auch initialisiert werden. Der Variablen folgen dabei ein Gleichheitszeichen und ein Ausdruck. Im Ausdruck dürfen nur Variablen verwendet werden, die bereits definiert sind. Der Ausdruck darf in geschweifte Klammern eingeschlossen sein.

Beispiel:
static int a=20, b={(a+1)/2};

Variablen der Speicherklasse **static** und globale Variablen werden einmalig vom Compiler initialisiert. Variablen der Speicherklassen **register** und **auto** werden normalerweise nicht initialisiert. Manche Compiler lassen trotzdem eine Initialisierung zu, die bei jedem Blockeintritt wie eine Laufzeitanweisung ausgeführt wird.
Nicht initialisierte Variablen der Speicherklasse **static** und globale Variablen werden mit 0 initialisiert. Nicht initialisierte Variablen in den Speicherklassen **register** und **auto** haben einen beliebigen, in der Regel nicht reproduzierbaren Wert.
Ist das zu initialisierende Objekt ein Feld, dann besteht die Initialisierung aus einer Liste von Initialisierungen für die einzelnen Feldelemente. Die Liste ist in geschweifte Klammern einzuschließen, wobei die einzelnen Werte durch Komma getrennt werden. Die Initialisierung erfolgt in der Reihenfolge der Anordnung der Feldelemente. Wenn nicht genügend Werte angegeben werden, so wird der Rest des Feldes mit 0 initialisiert. Bei mehrdimensionalen Feldern kann die Initialisierung zeilenweise erfolgen, wobei jede Liste auch getrennt in geschweifte Klammern eingeschlossen werden kann. Felder der Speicherklasse **auto** können nicht initialisiert werden.

Beispiele:
```
static int a[10] = {1,2,5,7,10};
static int b[3] [4] = {{1,2,5,7},
                       {3,8,12,17},
                       {10,2,3,12}};
oder
static int b[3] [4] =
{1,2,5,7,3,8,12,17,10,2,3,12};
static int c[3] [4] =
{ {1}, {2}, {3} };
```

Das Feld **a** wird mit 5 Werten initialisiert. Die restlichen Elemente werden auf 0 gesetzt. Die zwei Schreibweisen für die Initialisierung des Feldes **b** sind identisch. Im Feld **c** wird nur die erste Spalte initialisiert. Die restlichen Feldelemente werden auf 0 gesetzt.
Bei der Initialisierung von Feldern ist die Dimensionsangabe nicht zwingend er-

forderlich. Die Dimensionsangabe wird dann aus der Anzahl der Werte der Initialisierungsliste berechnet. Von dieser Möglichkeit wird auch bei der Initialisierung eines Zeichenfeldes mit einer Zeichenkette Gebrauch gemacht. Die geschweiften Klammern können bei der Initialisierung mit einer Zeichenkette entfallen.

Beispiel:
int d[] = {3,7,9,11};
char text[] = "Montag";
oder
char text [] =
{'M', 'o', 'n', 't', 'a', 'g', '\0'};

Die Verwaltung von mehreren Zeichenketten in einem Feld erfolgt zweckmäßigerweise in einem Zeigerfeld, das mit den Adressen der Zeichenketten initialisiert werden kann.

Beispiel:
static char * tag = {
"",
"Montag",
"Dienstag",
"Mittwoch",
"Donnerstag",
"Freitag",
"Sonnabend",
"Sonntag"
};

7. Strukturen und Vereinigungen

Eine Struktur (**struct**) ist ein Datenobjekt aus einer Folge von einzelnen benannten Komponenten, die Datenobjekte verschiedener Datentypen repräsentieren können. Der Speicherbereich einer Struktur wird vom Compiler aus der Summe der Speichergrößen der einzelnen Komponenten berechnet.
Eine Vereinigung (**union**) ist dagegen ein Datenobjekt, das auf einen virtuellen Speicherbereich Komponenten verschiedener Datentypen vereinigt. Die verschiedenen Komponenten können jedoch den Speicherbereich nur nacheinander belegen. Der Speicherbereich einer Vereinigung wird vom Compiler der größten Komponente entsprechend festgelegt.
Strukturen und Vereinigungen werden analog bereinbart und benutzt. In Strukturen und Vereinigungen können weitere Strukturen und Vereinigungen enthalten sein.
Strukturen und Vereinigungen werden wie folgt definiert:

struct [*name*] [{*member* [*member*...]}] [*identifier* [,*identifier*...]];
union [*name*] [{*member* [*member*...]}] [*identifier* [,*identifier*...]];

wobei
name – wahlfreier Name zur Bezeichnung der Struktur oder Vereinigung (Strukturname oder Vereinigungsname),
member – Spezifikation einer oder mehrerer Variablen eines bestimmten Datentyps und
identifier – Identifikator für eine Strukturvariable oder Vereinigungsvariable darstellen.

7.1. Definition von Strukturen und Vereinigungen

Werden in einer Struktur- bzw. Vereinigungsvereinbarung keine Identifikatoren angegeben, so wird lediglich die Struktur bzw. Vereinigung definiert. Es wird ein Name festgelegt, über den auf die Struktur- bzw. Vereinigungsvereinbarung Bezug genommen werden kann. Es erfolgt keine Vereinbarung von Struktur- bzw. Vereinigungsvariablen und somit auch keine Speicherplatzreservierung.

Beispiele:
struct art {char * nr; float pr;
int n;};
struct datum {int tag; char *
monat, jahr [2]};

Die erste Vereinbarung definiert eine Struktur **art** zur Beschreibung eines Artikels, bestehend aus einem Zeiger auf eine Zeichenkette zur Angabe einer Artikelnummer, einer **float**-Variablen für den Preis und einer **int**-Variablen für eine Stückzahl. Die zweite Vereinbarung definiert eine Struktur **datum**, die drei Komponenten enthält; eine **int**-Variable **tag**, einen Zeichenkettenzeiger **monat** und ein Zeichenfeld **jahr** aus zwei Zeichen.
Über die Namen **art** und **datum** kann im weiteren auf die definierten Strukturen Bezug genommen werden.

7.2. Vereinbarungen von Struktur- und Vereinigungsvariablen

Werden nach der Struktur- bzw. Vereinigungsvereinbarung Identifikatoren spezifiziert, so bezeichnen diese Identifikatoren im weiteren Struktur- bzw. Vereinigungsvariablen, deren Speicherobjekt die entsprechende Struktur bzw. Vereinigung realisiert.

Beispiele:
struct {char * nr; float pr;
int n;} a, b;
struct {int tag; char * monat,
jahr [2]} d1, d2, d3;

Es werden zwei Strukturvariablen **a** und **b** bzw. drei Strukturvariablen **d1**, **d2** und **d3** vereinbart, die jeweils ein Speicherobjekt der angegebenen Strukturen repräsentieren. Die Strukturen selbst sind unbenannt. Die Speicherstruktur der Strukturvariablen **a** ist in Bild 2 dargestellt.

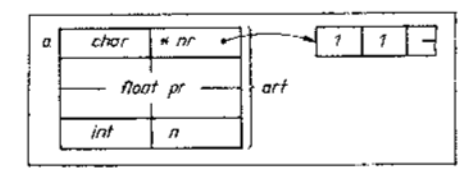


Bild 2 Beispiel für den Aufbau einer Struktur

Falls die Struktur bzw. Vereinigung bereits gemäß 7.1. definiert wurde, so kann die Vereinbarung der Struktur- bzw. Vereinigungsvariablen durch Bezugnahme auf den Namen der Struktur bzw. Vereinigung vereinfacht werden.

Beispiele:
struct datum d1, d2, d3;

Es werden drei Strukturvariablen **d1**, **d2** und **d3** für die Struktur **datum** vereinbart.

Entsprechend den Abschnitten 5. und 6. können auch Felder von Strukturen und Vereinigungen bzw. Zeiger auf Strukturen und Vereinigungen vereinbart werden.

Beispiele:
struct art a, b, t[20], * p;
struct datum d1, d2, d3,
tlist [200], * t1;

Das Feld **t** besteht aus 20 Feldelementen, wobei jedes Element die Struktur **art** realisiert.
Das Feld **tlist** besteht aus 200 Feldelementen, wobei jedes Element eine Struktur **datum** verkörpert.

Programmierung in C

(Teil IV)

Dr. Thomas Horn
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

Im vierten Teil dieser Artikelreihe soll nun mit der Benutzung der Struktur- und Vereinigungsvariablen sowie der Initialisierung ihrer Komponenten die Behandlung der Strukturen und Vereinigungen abgeschlossen werden. Der folgende Abschnitt wird dann das bereits im ersten Teil vermittelte Grundwissen über die Funktionen durch die Behandlung des Parameterkonzepts vertiefen und auf spezielle Probleme der Anwendung von Funktionen, wie Zeiger auf Funktionen und rekursiver Funktionsaufruf, eingehen.
Der letzte Abschnitt dieses Teils erläutert die Funktionen und die Anwendung des Makro-Präprozessors, der insbesondere für die Rationalisierung der Programmentwicklung größerer Softwaresysteme und für die Systemprogrammierung von Bedeutung ist.

7.3. Benutzung von Struktur- und Vereinigungsvariablen und deren Komponenten

In C sind mit Strukturen und Vereinigungen keine Operationen außer der Bildung von Adressen und der Adreßübergabe in Funktionen möglich.

Beispiel:
p=&a; p=t; p=&t[i];

Alle weiteren Operationen im Zusammenhang mit Strukturen und Vereinigungen beziehen sich nur auf ihre Komponenten und auf Zeiger.
Auf die einzelnen Komponenten einer Struktur oder Vereinigung kann entweder über den Variablennamen der Struktur bzw. Vereinigung oder über einen Zeiger auf die Struktur bzw. Vereinigung Bezug genommen werden.
Bei Verwendung des Variablennamens der Struktur oder Vereinigung wird der Komponentenname über einen Punkt (.) als Kettungsoperator für Komponenten angehangen.

Beispiele:
a.nr, a.pr, a.n
tlist[10].tag, tlist[10].monat,
tlist[10].jahr
tlist[i].tag, tlist[i].monat,
tlist[i].jahr

Bei Verwendung eines Struktur- bzw. Vereinigungszeigers wird über den Operator → auf die Komponenten der Struktur zugegriffen.

Beispiele:
p→nr, p→pr, p→n
tl→tag, tl→monat, tl→jahr

Die Schreibweise **p→nr** ist der Schreibweise (*p).nr äquivalent usw.
Bei der Anwendung der Inkrement- bzw. Dekrementoperatoren ist zu beachten, daß die Operatoren **.** und → eine höhere Priorität haben:

tl→tag++	Die Komponente **tag** wird inkrementiert.
++tl→tag	Die Komponente **tag** wird inkrementiert.
(tl++)→tag	Nach dem Zugriff auf die Komponente **tag** wird **tl** inkrementiert (**tl** zeigt auf die nächste Struktur).
(++tl)→tag	Vor dem Zugriff auf die Komponente **tag** wird **tl** inkrementiert.

Beispiel:
Eine Vereinigung **tab** vereinigt auf dem gleichen Speicherbereich ein Zeichenfeld **buf** mit 512 Zeichen und 64 Strukturen **art** für Artikel:
union tab {char buf [512];
struct art e [64];};

Durch die folgende Anweisung werden 10 Vereinigungen als Vereinigungsfeld **a** definiert:

union a[10], *pa;

Der Zeiger **pa** ist ein Vereinigungszeiger. Bild 4 zeigt den Aufbau der Vereinigung.
Über **a[i].buf[j]** können die Bytes (Zeichen) der Vereinigung angesprochen werden, wobei **i** die Vereinigung und **j** das Byte innerhalb der Vereinigung adressieren. Die Strukturen **art** mit ihren Komponenten können über

a[i].e[k].nr, a[i].e[k].pr und **a[i].e[k].n**

angesprochen werden, wobei **k** die Nummer der Struktur innerhalb der Vereinigung **i** angibt.

7.4. Initialisierung von Strukturen und Vereinigungen

Die Initialisierung von Strukturen ist nur für die Speicherklasse **static** bzw. für globale Strukturen möglich. Strukturen in der Speicherklasse **auto** und Vereinigungen können prinzipiell nicht initialisiert werden.
Bei der Initialisierung einer Struktur werden die verschiedenen Initialisierungswerte in einer Liste niedergeschrieben und in geschweifte Klammern eingeschlossen. Bei der Initialisierung eines Strukturfeldes sollte jedes Feldelement gesondert in geschweifte Klammern eingeschlossen werden.

Beispiele:
static struct datum a =
{22,„April“, {‚8‘, ‚5‘}};
static struct art t[]=
{„11–001“, 2.5, 10},
{„11–015“, 5.25, 120}
};

7.5. Neue Typbezeichner

Durch Anwendung der **typedef**-Anweisung können auch Strukturen- bzw. Vereinigungstypbezeichner definiert werden.

Beispiel:
typedef struct {int tag;
char *monat, jahr[2]} date;
oder
typedef struct datum date;

In beiden Anweisungen wird der neue Typbezeichner **date** vereinbart, der zur Deklaration von Variablen für die Struktur **datum** benutzt werden kann:
date a,b,c;

7.6. Rekursive Strukturen

Rekursive Strukturen sind Strukturen, die Zeiger auf eine gleichartige Struktur enthalten. Die Struktur darf sich jedoch selbst nicht enthalten, aber Zeiger sind erlaubt. Unberührt davon bleibt, daß eine Struktur beliebige andere Strukturen, die vor ihr definiert wurden, in sich enthalten darf. Die Bezugnahme auf andere Strukturen ist kein rekursives Sprachelement.

Beispiel:
Es soll eine Struktur aufgebaut werden, die es gestattet, verschiedene Ereignisse miteinander zu verketten. Die Struktur soll enthalten:
- das Datum des Ereignisses,
- die Adresse einer Zeichenkette, die das Ereignis spezifiziert,
- einen Zeiger auf das vorhergehende Ereignis und
- einen Zeiger auf das nachfolgende Ereignis.

```
struct entity {struct datum TAG;
    char *text;
    struct entity *vorher;
    struct entity *nachher;};
struct entity a,b,c,d,*i;
```

Zugriff auf die Strukturelemente der Struktur **a**:

```
a.TAG.tag
a.TAG.monat
a.TAG.jahr
a.text
a.vorher
a.nachher
```

Zugriff auf die Strukturelemente des Nachfolgers:

```
i=a.nachher
i→TAG.tag
i→TAG.monat
i→TAG.jahr
i→vorher
i→nachher
```

7.7. Bitfelder

Auf der Grundlage der Strukturen können auch Bitfelder vereinbart werden. Bitfelder sind immer vom Typ **unsigned.** Die Bitanzahl darf ein Integerwort nicht überschreiten (maximal 16 Bits). Die Bitanzahl wird nach dem Komponentennamen über Doppelpunkt getrennt spezifiziert. Für Zwischenräume können auch unbekannte Bitfelder benutzt werden. Die Bitanzahl 0 kann zur Ausrichtung auf das nächste Integerwort verwendet werden. Die Bits werden in den meisten Rechnersystemen von rechts nach links im Integerwort angeordnet.

Beispiel:
Definition der Struktur eines Steuerregisters für ein peripheres Gerät:

```
struct dk {
    unsigned go:1;
    unsigned function:3;
    unsigned adrext:2;
    unsigned ready:1;
    unsigned trap:1;
    unsigned:3;
    unsigned busy:1;
    unsigned errcode:3;
    unsigned error:1;
    } ctrldk1;
```

Auf die Bitfelder kann wie in Strukturen üblich zugegriffen werden. Bitfelder funktionieren wie kleine ganze Zahlen ohne Vorzeichen und können wie andere Werte in Ausdrücken verwendet werden.

Beispiele:

```
if (ctrldk1.error==1) printf („...");
ctrldk1.function=2;
/* Function read */
crtldk1.go=1;
/* Start controller */
```

Hinweise:

1. Bitfelder haben keine Vorzeichen.
2. Es gibt keine (indizierten) Felder von Bitfeldern.
3. Bitfelder haben keine Adressen.

7.8. Aufzählungstypen

In neueren Implementierungen gibt es Aufzählungstypen, bei denen wie in PASCAL die möglichen Werte explizit aufgeführt werden. Den aufgezählten Werten werden intern die Integerzahlen ab 0 beginnend zugewiesen. Die Vereinbarungen der Aufzählungstypen und der Variablen für Aufzählungstypen werden ähnlich wie in Strukturen über das Schlüsselwort **enum** vereinbart.

```
enum Farbe {weiß, schwarz, rot,
gelb, grün, blau};
enum Farbe f1, f2;
```

Die Variablen **f1** und **f2** sind vom Aufzählungstyp **Farbe** und können die oben spezifizierten Werte annehmen. Mit Variablen vom Aufzählungstyp können alle Operationen ähnlich wie mit Variablen vom **int**-Typ ausgeführt werden, wie Zuweisungen, Vergleiche usw.

Beispiele:

```
f1=weiß; f2=rot;
if (f1==gelb) f1=blau;
```

Hinweis:

Aufzählungstypen werden insbesondere vorteilhaft für die Definition der Werte von Bitfeldern verwendet, was die Programmierung von Operationen mit Bitfeldern wesentlich vereinfachen kann.

8. Funktionen

Größere Problemstellungen werden mit Hilfe von Funktionen in kleinere zerlegt. Diese Vorgehensweise gestattet die Entwicklung modularer, gut strukturierter Programmsysteme. Häufig benötigte, universell nutzbare Funktionen können in Objektmodulbibliotheken katalogisiert und in andere Programmsysteme eingebunden werden. Damit wird das Programmsystem im Ganzen klarer, und eine mehrfache Entwicklung von Funktionsmoduln wird vermieden.

8.1. Definition von Funktionen

In C ist jedes Programm grundsätzlich eine Funktion, die von anderen Funktionen aufgerufen werden kann. In einer Übersetzungseinheit können mehrere Funktionen definiert werden, jedoch ist die Definition von Funktionen innerhalb von Funktionen nicht zulässig. Jede Funktion kann auch separat übersetzt werden.
Eine Funktion hat folgenden prinzipiellen Aufbau:

type name(liste_der_formalen_parameter)
deklaration_der_formalen_parameter;
{
funktionsblock
}

Der Name *name* legt den Funktionsnamen fest. In runden Klammern erfolgt die Spezifikation der formalen Parameter, falls die Funktion formale Parameter hat. Nach der Parameterliste erfolgt die Typenvereinbarung der formalen Parameter. Nach der Vereinbarung der formalen Parameter folgt der Funktionsblock, der in geschweiften Klammern Vereinbarungen und Anweisungen enthält. Die Funktion wird verlassen, das heißt, die Steuerung wird an die aufrufende Funktion zurückgegeben, wenn die rechte geschweifte Klammer erreicht wird.
Mit der **return**-Anweisung kann die Funktion vorzeitig verlassen werden. Wird in der **return**-Anweisung ein Ausdruck spezifiziert, so wird der Wert des Ausdrucks als Funktionswert an die aufrufende Funktion übermittelt. Der Typ des Funktionswertes wird durch die Typangabe *type* vor dem Funktionsnamen festgelegt. Bei ausgelassener Typangabe wird **int** angenommen.
Wird bei einer Funktion keine Speicherklasse spezifiziert, so gilt der Funktionsname als global vereinbart. Wird die Speicherklasse **static** spezifiziert, so kann auf die Funktion nur von anderen Funktionen innerhalb des Übersetzungsmoduls zugegriffen werden.
Im allgemeinen können die Funktionsnamen frei gewählt werden. Da Funktionsnamen aber in der Regel globale Symbole sind, sollten die Einschränkungen des Basisbetriebssystems berücksichtigt werden, zum Beispiel läßt das Betriebssystem OS-RW nur globale Symbole aus maximal 6 Zeichen zu, wo-

bei nicht zwischen Klein- und Großbuchstaben unterschieden wird. Weiterhin hat der Funktionsname **main** eine besondere Bedeutung; er legt die Startadresse des Programmsystems (der Task)fest. Aus diesem Grund muß die oberste Funktion, die durch das Betriebssystem gestartet werden soll, immer **main** heißen.

Beispiele:
leer(){}
Eine leere Funktion, die nichts bewirkt.

```
main()
{int i;
for(i=1;i<=10;i++)
printf(„\ni=%d j=%d\n“,i,i*i);
}
```

Die Funktion kann durch das Betriebssystem gestartet werden. Sie erzeugt eine Tabelle der Quadrate der Zahlen von 1 bis 10.

```
float factor(x)
int x;
{
float y=1.0;
while (x>1) {y*=x; x−=1;}
return(y)
}
```

Funktion zur Berechnung der Fakultät der Zahl x. Der Funktionswert hat den Typ **float,** unabhängig vom Typ des Ausdrucks in der **return**-Anweisung.

8.2. Aufruf von Funktionen

Funktionen werden über ihren Namen und die Angabe einer Liste von aktuellen Argumenten in runden Klammern aufgerufen:

name(liste_der_aktuellen_argumente)

Wenn die Funktion parameterlos ist, müssen zumindest die runden Klammern angegeben werden, da der Compiler an den runden Klammern den Funktionsaufruf erkennt. Die Liste der aktuellen Argumente muß entsprechend der Liste der formalen Parameter aufgebaut sein (Stellungsparameter!). Einfache Variablen werden in C nach der Methode *„call by value“* übergeben, das heißt, es wird der Wert des entsprechenden Arguments des Funktionsaufrufes berechnet und der Variablen des formalen Parameters zugewiesen. Damit arbeitet die aufgerufene Funktion mit einer lokalen, temporären Kopie eines jeden Arguments. Das bedeutet, daß eine Funktion das ursprüngliche Argument der aufrufenden Funktion nicht verändern kann. Anders ist es bei Feldern, Strukturen, Vereinigungen und Funktionen als Argumente einer Funktion. Bei diesen Datentypen wird die Anfangsadresse des Datenobjekts übergeben, das heißt, das Datenobjekt wird nicht kopiert, sondern die aufgerufene Funktion greift unmittelbar über die übergebene Adresse auf die Datenobjekte zu. Diese Methode der Parameterübermittlung wird als *„call by reference“* bezeichnet.
Bei einem Feld werden somit die Feldelemente nicht kopiert, und es wird in der aufgerufenen Funktion keine Kopie des Feldes angelegt. Die aufgerufene Funktion kann die Feldelemente problemlos ändern. Ähnlich wird auch bei Strukturen, Vereinigungen und Funktionen verfahren. Sollen einfache Variablen in der aufrufenden Funktion verändert werden, so muß entsprechend die explizite Übergabe der Adresse der Variablen vereinbart und in der Funktion mit den Zeigern auf die Variablen gearbeitet werden.
Ein weiteres Problem ist die Typvereinbarung für den Funktionswert, der an die aufrufende Funktion übermittelt wird. Da Funktionen immer externe (bzw. **static-**) Funktionen in bezug auf die aufrufende Funktion sind, kann die aufrufende Funktion den Typ des Funktionswertes nicht kennen. Für nichtspezifizierte aufgerufene Funktionen wird standardmäßig immer **int** angenommen. Werden Werte anderer Typen übermittelt, so muß die Funktion spezifiziert werden, zum Beispiel kann die Funktion **factor** aus Pkt. 8.1. wie folgt spezifiziert werden:

float factor();

Damit ist festgelegt, daß **factor** eine (externe) Funktion ist und der Funktionswert den Typ **float** hat. Bei Funktionen, die durch den Taskbuilder eingebunden werden, kann keine Typwandlung mehr vorgenommen werden, so daß bei nicht richtiger Spezifikation des Funktionswertes schwerwiegende Fehler entstehen können.

Beispiel:
Es sei eine Struktur **datum** gegeben. In **main** sei ein Strukturenfeld **tlist** vorhanden. Es soll eine Funktion **search** definiert werden, die eine Struktur mit einem bestimmten Monatsnamen sucht und den Zeiger als Funktionswert übergibt.

```
struct datum{int tag; char*monat,
jahr[4]};
main()
{
struct datum tlist[100],
*search(),*i;
...
i=search(tlist,„April“);
...
}
struct datum *search(l,m)
struct datum *l[]; char*m[];
{
int i;
for(;;l++)
for(i=0; l→monat[i]==m[i]; i++)
    if(m[i]==‚\o‘) return(l);
}
```

8.3. Argumente aus der Kommandozeile

Im Betriebssystem installierte Tasks können mit Angabe einer Kommandozeile, bestehend aus mehreren Argumenten, aufgerufen werden, zum Beispiel

tsk arg1 arg2 ...

Diese Kommandozeile kann in der Funktion **main** ausgewertet werden. Zwei Argumente sind dafür in **main** zulässig:
argc – die Anzahl der Argumente in der Kommandozeile,
argv – Zeichenkettenfeld mit den Argumenten aus der Kommandozeile.

Beispiel:

```
main(argc,argv)
int argc; char *argv[];
{
...}
```

Nach der Betriebssystemkonvention ist **argv[0]** die Adresse des Namens, unter dem das Programm aufgerufen wurde (*tsk*). **argv[1]** ist die Adresse der Zeichenkette des ersten Arguments *arg1* usw. Wenn die Task ohne zusätzliche Argumente gestartet wurde, ist **argc** zumindest gleich 1.

8.4. Zeiger auf Funktionen

In C ist es möglich, einen Zeiger auf eine Funktion zu definieren, der an Funktionen übergeben oder in Tabellen gespeichert werden kann usw.

Beispiel:
Angenommen, es sei eine Funktion **swap (a,b)** definiert, die die **float**-Werte miteinander vertauscht und den größeren Wert als Funktionswert übermittelt.
Mit
float swap();
kann die Funktion deklariert werden.
Mit
float (*funct)();

kann ein Zeiger auf eine Funktion vereinbart werden, die einen Funktionswert vom Typ **float** als Resultat liefert. Die Adresse der Funktion **swap** kann wie folgt gebildet werden:
funct=(&swap)();
Eine Funktion kann über einen Zeiger wie folgt gestartet werden:
f0=(*funct)(tab[i], tab[i+1]);

Wird eine Funktion als Parameter einer anderen Funktion übergeben, so wird stets die Adresse der Funktion übergeben. Der Adreßoperator ist in diesem Fall unnötig.

Beispiel:
```
main()
{
float swap(), sort();
...
sort (a,b,swap)
...
}
float sort (x,y,change)
float x[],y[],(*change)();
{
...
(*change)(x[i],y[j]);
...
}
```

8.5. Rekursiver Funktionsaufruf

In C können Funktionen rekursiv benutzt werden, d. h. daß eine Funktion sich selbst entweder direkt oder indirekt aufrufen kann.

Beispiel:
Berechnung der Fakultät einer Zahl x:
```
float factor(x)
int x
{
if (x==1) return (1.0);
   else return (x*factor(x-1));
}
```

Rekursionen sparen im allgemeinen keinen Speicherplatz, da im Stack eine Reihe von Zwischenresultaten und Adressen abgelegt werden muß. Der Stack wird dadurch sogar erheblich belastet. Ebenso sind Rekursionen nicht schneller, aber rekursive Lösungen sind in Abhängigkeit vom Problem (rekursiv definierte Datenstrukturen usw.) oft kompakter, leichter zu schreiben und zu verstehen.

Hinweis:
In Abhängigkeit vom Grad der Nutzung von rekursiven Funktionsaufrufen (Rekursionstiefe) ist der Stackbereich zu vergrößern (z. B. im Betriebssystem OS–RW mit dem Taskbuilder TKB).

9. Der Makro-Präprozessor

Der C-Compiler verfügt über einen Makro-Präprozessor, der einige Spracherweiterungen wie Fileeinfügungen, Makrosubstitutionen und bedingte Generierungsanweisungen realisiert.
Der Makropräprozessor generiert in einem Vorpaß das eigentliche Quellprogramm, das anschließend übersetzt wird. Die Anweisungen an den Makro-Präprozessor sind gegenüber den C-Sprachanweisungen zeilengebunden und beginnen mit einem Doppelkreuz (#) in der Position 1. Sie können an beliebiger Stelle im Quelltext auftreten, unabhängig von der Quellprogrammstruktur. Ihre Syntax ist nicht mit den Sprachelementen von C gekoppelt. Die Anweisungen an den Makro-Präprozessor belegen grundsätzlich immer eine ganze Zeile. Falls aber eine Zeile, zum Beispiel bei Makrodefinitionen, nicht ausreichend ist, können Fortsetzungszeilen verwendet werden, wenn die vorhergehende Zeile als letztes Zeichen einen Backslash (\) enthält.

9.1. Fileeinfügungen

#include "*filename***"**
fügt ein File aus dem aktuellen Fileverzeichnis des Nutzers in das C-Programm ein, während

#include<*filename*>

ein File aus dem Bibliotheksfileverzeichnis (im OS–RW: LB:[1,1]) einfügt. Es kann ein beliebiger C-Text eingefügt werden, insbesondere Strukturbeschreibungen, Makrodefinitionen usw. Die **#include**-Anweisung kann geschachtelt angewendet werden, das heißt, in einem eingefügten File dürfen weitere **#include**-Anweisungen benutzt werden.

9.2. Makrodefinitionen und -substitutionen

#define *identifier string*
ist eine Makrodefinition ohne Argumente und dient der einfachen Zeichenkettensetzung; zum Beispiel ersetzt

#define EOF (−1)

im weiteren C-Text alle Zeichenketten **EOF** durch **(−1).**

#define *identifier*(*identifier*, . . .) *string*

ist eine Makrodefinition mit Parametern. Bei Makroaufruf werden im Makrokörper *string* alle formalen Parameter durch aktuelle Parameter ersetzt.

Beispiel:
```
#define min (x,y) ((x)<(y)? (x):(y))
...
Amin=min(A[i],A[i+1]);
```
Infolge der Makrogenerierung wird folgender C-Text übersetzt:
```
Amin=((A[i])(A[i+1])?
(A[i]):(A[i+1]));
```

9.3. Streichen von Makrodefinitionen

#undef *identifier*
streicht die angegebene Makrodefinition aus dem Verzeichnis des Makro-Präprozessors. Damit ist der Identifikator im weiteren undefiniert.

Beispiel:
```
#undef min
#undef EOF
```
Die Makrodefinitionen **min** und **EOF** werden gestrichen.

9.4. Bedingte Compilierung

Ein bedingter Anweisungsblock wird durch eine der folgenden drei Anweisungen eingeleitet:

#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*

Die nachfolgenden C-Anweisungen werden generiert, wenn die Bedingung **TRUE** bzw. der Konstantenausdruck ungleich 0 ist. Abgeschlossen wird der bedingte Anweisungsblock durch eine Anweisung

#endif

Ist zwischen der **#if**-Anweisung und der **#endif**-Anweisung eine Anweisung

#else

enthalten, so wird die Generierungsbedingung umgekehrt. Damit werden entweder die Anweisungen bis zur **#else**-Anweisung oder danach generiert. Bedingte Anweisungsblöcke können ineinander geschachtelt werden.

Die Reihe zu C wird im nächsten Heft fortgesetzt mit Ausführungen zu den Ein- und Ausgabefunktionen und zur Methodik der Progammentwicklung.

Programmierung in C

Teil V

Dr. Thomas Horn
Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden

Im vorletzten Teil der Artikelreihe zu C wird das Ein- und Ausgabekonzept der Programmiersprache C behandelt, das kein integrierter Bestandteil der Sprachdefinition ist, sondern über eine umfangreiche Funktionsbibliothek realisiert wird. Es gibt ein Minimum an E/A-Funktionen, die als Quasi-Standard von allen C-Systemen realisiert werden. Diese Funktionen werden im wesentlichen im vorliegenden Beitrag behandelt. Sie garantieren auch eine Portabilität der C-Programme unter den verschiedenen Rechnern und Betriebssystemen. Weiterhin werden einige Hinweise zur Verarbeitung von C-Programmen gegeben. Das Prinzip wird hier im wesentlichen allgemeingültig dargestellt, wobei unter den einzelnen Betriebssystemen im Detail, vor allem bei den Kommandos und Filespezifikationen, geringfügige Abweichungen auftreten können. Über diese Besonderheiten sollte man sich immer informieren, wenn man mit einem bestimmten C-Sprachverarbeitungssystem unter einem konkreten Betriebssystem zu arbeiten beginnt. Der letzte Teil dieser Artikelreihe wird dann einige komplexe Programmbeispiele behandeln, an denen auch die Programmierungstechnik in C verdeutlicht werden soll.

9.5. Zeilennumerierung

#line *constant identifier*

definiert die Zeilennummer und den Namen eines Quellprogramms. Die Präprozessoranweisung wird zur besseren Identifikation von Fehlermeldungen bei C-Programmen benutzt, die durch zahlreiche Fileeinfügungen unübersichtlich geworden sind. Die Anweisung ermöglicht, daß sich die Fehlermeldungen auf die Zeilennummern verschiedener Quellfiles beziehen können. Wenn in der Anweisung kein Identifikator spezifiziert ist, so wird der letzte Identifikator weiter benutzt.

Beispiel:
#line 1 TESTO
Die folgende Zeile des Files **TESTO** bekommt die Zeilennummer 1.

Beispiel:
Wenn der Generierungsparameter **TIME** ungleich 0 ist, soll die Struktur **datum** mit der Komponente Zeit generiert werden, anderenfalls soll die Struktur keine Komponente Zeit definieren.

```
#define TIME 1
...
#if TIME
struct datum
{int tag;
char *monat;
char jahr[2];
int zeit[2];
#else
struct datum
{int tag;
char *monat;
char jahr[2];}
#endif
```

Die generierungsabhängige Struktur **datum** kann auch vereinfacht wie folgt definiert werden:

```
struct datum
{int tag;
char *monat;
char jahr[2];
#if TIME
int zeit[2];
#endif
}
```

Die zweite Schreibweise ist der ersten äquivalent.

10. Ein- und Ausgabefunktionen

Die Ein- und Ausgabeorganisation ist, wie bereits erwähnt, kein Bestandteil der Sprachdefinition der Programmiersprache C. Sie wird grundsätzlich über Funktionen der Standard-Unterprogrammbibliothek (CSLIB.OLB oder C.OLB) realisiert, die größtenteils Assemblerunterprogramme zur Anpassung an das Betriebssystem des jeweiligen Rechners enthält. Durch die Standard-Unterprogramme der Unterprogrammbibliothek wird im wesentlichen die Portabilität der C-Programme abgesichert. Diese Unterprogramme (Funktionen) werden nachfolgend behandelt.

10.1. Prinzipien der Fileverarbeitung

Unter einem File wird eine logisch zusammenhängende Menge von Datensätzen verstanden, die nach einem bestimmten Organisationsprinzip auf externen Datenträgern, wie Disketten, Plattenspeicher, Magnetbandkassetten usw., gespeichert werden. Vielfach wird in der Literatur der früher stärker benutzte Begriff Datei dem Begriff File gleichgesetzt, wobei aber der Begriff Datei vom Organisationsprinzip des Files (Filestruktur) abstrahiert. Files werden auf den externen Datenträgern durch Filespezifikationen identifiziert. Eine Filespezifikation besteht im allgemeinen aus einem Gerätenamen (*dev*:), einem Verzeichnisnamen (*dir*/oder **[***dir***]**), einem Filenamen (*fname*) und einem Filetyp (*.ftyp*):

dev:*dir*/*fname.ftyp*
oder
dev:**[***dir***]***fname.ftyp*;*version*

In CP/M-kompatiblen Betriebssystemen entfällt generell die Angabe eines Verzeichnisses. In MS-DOS- bzw. UNIX-kompatiblen Betriebssystemen wird das Verzeichnis durch einen Schrägstrich vom Filenamen getrennt. Es können auch – durch Schrägstriche getrennt – noch Unterverzeichnisse (*Subdirectories*) spezifiziert werden. Im Betriebssystem OS-RW auf SKR-Anlagen wird das Verzeichnis generell in eckigen Klammern spezifiziert, und nach dem Filetyp kann noch eine Versionsangabe (;*version*) folgen. Beim Zugriff auf das Standardgerät, das Standardverzeichnis des Nutzers und die letzte Versionsnummer eines Files können die entsprechenden Angaben in der Filespezifikation entfallen, so daß nur der Filename und der Filetyp angegeben werden müssen.
Für nicht filestrukturierte Geräte, wie Drucker, Terminals usw., müssen die entsprechenden Gerätenamen spezifiziert werden, wobei die Angabe von Filename und Filetyp entfallen kann, z. B. für den Drucker in Abhängigkeit vom Betriebssystem **PRT:**, **LST:** oder **LP:** usw.
In Betriebssystemen werden sequentielle Files und Direktzugriffsfiles unterschieden.
Sequentielle Files können auf allen Geräten ein- oder ausgegeben werden, wie Diskettenspeicher, Magnetbandgeräte,

Terminals, Drucker usw. Direktzugriffsfiles können nur auf Geräten mit wahlfreiem Zugriff, wie Diskettenspeicher und Plattenspeicher, ein- und ausgegeben werden. Da die sequentiellen Files von grundlegender Bedeutung sind und betriebssystemunabhängig realisiert werden können, „standardisiert" die Standard-Unterprogrammbibliothek nur die Verarbeitung sequentieller Files. Die Verarbeitung von Direktzugriffsfiles ist im wesentlichen betriebssystemabhängig und soll deshalb nur als Beispiel für das Betriebssystem UNIX zum Schluß dieses Abschnitts skizziert werden.
Die Verarbeitung von Files schließt folgende Funktionen ein:
- Eröffnen des Files (*open*)
- Ein- oder Ausgabe der Datensätze (*read* oder *write* bzw. *get* oder *put*)
- Schließen des Files (*close*).

Für die Fileeröffnung wird bei sequentiellen Files die Funktion **fopen** verwendet. Bei der Fileeröffnung wird vom Laufzeitsystem ein Filedeskriptorblock (FDB) eingerichtet. Bei der Fileverarbeitung mit den Ein- und Ausgabefunktionen, wie **getc**, **putc**, **fgets**, **fputs** usw., wird auf den FDB über einen Filepointer *fp* Bezug genommen. Nach der Fileverarbeitung wird mit der Funktion **fclose** das File wieder geschlossen und der FDB gelöscht.
Zur Vereinfachung der Anwendung der Fileverarbeitung existiert ein File *STDIO.H*, das die Strukturdefinition für den FDB enthält. Die Bezugnahme erfolgt über den Strukturentyp **FILE**, mit dem beliebige Filepointer eingerichtet werden können. Voraussetzung ist, daß das File *STDIO.H* mit der Anweisung **#include <stdio.h>** am Programmanfang in das C-Programm eingefügt wurde. (Das File *STDIO.H* enthält außerdem noch Definitionen für die Symbole **FALSE** (0), **TRUE** (1), **NULL** (0), **EOF** (−1) und **EOS** (\0).)
Die Fileeröffnung kann dann wie folgt durchgeführt werden:

fp=**fopen (***name*, *mode***);**

wobei
name
die Filespezifikation angibt, die entsprechend den Festlegungen des verwendeten Betriebssystems aufgebaut sein muß,
mode
die Zugriffsart spezifiziert:
„**r**" – Lesezugriff,
„**w**" – Schreibzugriff,
„**a**" – Schreibzugriff zum Anfügen an das File.

Dem Filepointer wird bei fehlerfreier Fileeröffnung die Adresse des FDB zugewiesen. Bei fehlerhafter Fileeröffnung wird der Filepointer *fp* auf Null gesetzt. Diese Bedingung kann mit dem Symbol **NULL** getestet werden.
Der Fileabschluß erfolgt mit:

fclose (*fp***);**

Bei normaler Programmbeendigung wird jedes noch offene File automatisch geschlossen.
Durch das Laufzeitsystem werden drei Filepointer für das Standardgerät des Nutzers, das Bedienterminal (**CON:** oder **TI:**), automatisch eingerichtet:

stdin – als Standardeingabegerät
stdout – als Standardausgabegerät
stderr – als Standardfehlerausgabegerät.

Diese Filepointer brauchen nicht definiert und die Files nicht eröffnet werden (sind in *STDIO.H* definiert).

Beispiel:
```
#include <stdio.h>
main ()
{
FILE *fp1,*fp2;
if((fp1=fopen("TEST.DAT","r"))
        ==NULL) goto ENDE;
if((Fp2=fopen("LP:","w"))
        ==NULL) goto ENDE;
...
fclose(fp1); fclose(fp2);
ENDE:
}
```

10.2. Ein- und Ausgabe von Zeichen

Die zeichenweise Ein- und Ausgabe kann mit den Funktionen **getc** und **putc** durchgeführt werden:

c=**getc(***fp***);**
putc(*c*,*fp***);**

wobei *c* die Variable (im Byteformat) ist, die ein- oder ausgegeben werden soll. Wenn bei einer Eingabeanweisung kein Zeichen mehr zur Verfügung steht, so wird der Variablen *c* der Wert **−1** (**EOF** – **E**nd **O**f **F**ile) zugewiesen, der mit dem Symbol **EOF** ausgetestet werden kann.
Zur Vereinfachung der Nutzung des Standardein- bzw. -ausgabegerätes stehen zwei weitere Funktionen zur Verfügung: *c*=**getchar();**
entspricht: *c*=**getc(stdin);**
putchar(*c***);**
entspricht: **putc(***c***,stdout);**

Beispiel:
Angenommen, im Beispiel 10.1 soll das File **"TEST.DAT"** auf **"LP:"** protokolliert werden, so sind folgende Anweisungen zu ergänzen:

```
char c;
while((c=getc(fp1))!=EOF)
putc(c,fp2);
```

10.3. Ein- und Ausgabe von Zeilen

Die Ein- und Ausgabe von Zeilen kann mit den Funktionen **fgets** und **fputs** erfolgen:

s=**fgets(***line*,*lmax*,*fp***);**
fputs(*line*,*fp***);**

wobei
line das Feld mit der Zeile,
lmax die maximal einzugebende Zeilenlänge und
s Adresse des Satzes oder **NULL** (bei Fileende) ist.
Es ist zu beachten, daß das letzte Zeichen einer Zeichenkette immer ein Null-Zeichen (**EOS** – **E**nd **O**f **S**tring) ist; das heißt, wenn *lmax* mit 80 Zeichen vorgegeben ist, beträgt die tatsächliche maximale Zeilenlänge nur 79 Zeichen. Genauso fordert **fputs**, daß die auszugebende Zeile mit dem Steuerzeichen **EOS** abgeschlossen ist.
Zur Vereinfachung der Arbeit mit dem Standardgerät des Nutzers stehen zwei weitere Funktionen zur Verfügung:
gets(*line***)**
entspricht: **fgets(***line*,*lmax*, **stdin)**
puts(*line***)**
entspricht: **fputs(***line***,stdout)**

Beispiel
```
#include <stdio.h>
main()
{
char zeile[80]; FILE*fp1;
if((fp1=fopen("TEST.C","r"))
==NULL)
error("File can't be opened");
while(fgets(zeile,80,fp1))
fputs(zeile,stdout);
fclose(fp1);
}
```

10.4. Formatierte Ein- und Ausgabe

Für die formatierte Ein- und Ausgabe stehen die zwei Funktionen **fprintf** und **fscanf** zur Verfügung:

fprintf(*fp*,*format*,*arg1*,*arg2*, . . .**)**
fscanf(*fp*,*format*,*arg1*,*arg2*, . . .**)**

wobei
format – die Formatbeschreibung und
argi – die Argumente angeben.

Die Formatbeschreibung gibt den Aufbau des externen Datensatzes an, wobei spezifiziert wird, an welcher Stelle sich welche Argumente in welchem Datenformat befinden sollen. Die Datenformate der Argumente werden durch Formatelemente beschrieben, die mit ge-

wöhnlichen Textzeichen beliebig gemischt werden können. Bei der Ausgabe geben die Formatelemente somit an, an welcher Stelle welche Argumente in welchem Format eingefügt werden sollen. Bei der Eingabe geben die Formatelemente an, wie die Eingabezeichenkette zu interpretieren ist.
Die Formatelemente beginnen mit einem Prozentzeichen **%**. Das Format wird durch ein nachfolgendes Zeichen spezifiziert:

d dezimale Darstellung des Arguments (**int**)
u dezimale Darstellung des Arguments ohne Vorzeichen (nur Ausgabe) (**int**)
o oktale Darstellung des Arguments (**int**)
x hexadezimale Darstellung des Arguments (**int**)
c das Argument ist ein Zeichen (**char**)
s das Argument ist eine Zeichenkette (**char**)
f Eingabe eines Gleitkommaarguments (**float** oder **double**)
Ausgabe eines Gleitkommaarguments in der Punktdarstellung *[−]mmm.nnnnnn.*
e Ausgabe eines Gleitkommaarguments in der Exponentendarstellung *[−]m.nnnnnn***E***[+−]xx.*
g Ausgabe eines Gleitkommaarguments in der Form **%e** oder **%f**, je nachdem, welches Format kürzer ist.

Soll in dem Text ein %-Zeichen ausgegeben werden, so muß **%%** spezifiziert werden.
Bei der Ausgabe kann zwischen dem %-Zeichen und dem Formatzeichen eine ausführliche Druckformatspezifikation stehen:

– Ein *Minuszeichen*, das angibt, daß das umgewandelte Argument in seinem Feld linksbündig ausgerichtet wird.
– Eine *Zahl*, die die minimale Feldbreite definiert. Erfordert das Argument nicht die volle Feldbreite, so wird das Feld mit Leerzeichen aufgefüllt. Beginnt die Zahl, die die Feldbreite spezifiziert, mit einer Null, so wird das Feld mit Nullen aufgefüllt.
– Ein *Punkt* mit einer *Zahl*, die eine maximale Feldbreite definiert. Bei **float** und **double** gibt diese Zahl die Stellen nach dem Dezimalpunkt an.
– Ein *Buchstabe* **l**, der angibt, daß das zugehörige Argument im **long**-Format (bzw. **double**-Format) spezifiziert ist.

Bei der Eingabe werden Leerzeichen, Tabulatoren und Zeilenvorschubzeichen in der Formatbeschreibung ignoriert.
Sonstige Zeichen (außer **%**) müssen in der Eingabezeile auftreten. Die Formatelemente legen den Datentyp der Argumente fest. Zwischen dem **%**-Zeichen und dem Formatzeichen können stehen:
– Ein Stern (*), der angibt, daß das Eingabefeld übersprungen wird. Eine Wertzuweisung erfolgt nicht.
– eine Zahl, die die Feldbreite des Eingabefeldes spezifiziert
– ein Buchstabe **l** vor den Formatzeichen **d**, **o** oder **x** zeigt an, daß das entsprechende Argument im **long**-Format spezifiziert ist. Vor dem Umwandlungszeichen **e** oder **f** weist **l** auf das **double**-Format hin.

Bei der Eingabe ist zu beachten, daß die Argumente Zeiger sein müssen. Bei Feldvariablen ist das automatisch gegeben. Bei einfachen Variablen ist, falls kein Zeiger vorhanden, die Adresse über den Adreßoperator **&** zu spezifizieren.
Zur Vereinfachung der Nutzung des Standardein- und -ausgabegerätes stehen zwei weitere Funktionen zur Verfügung:

printf(*format,arg1,arg2,...***)**
scanf(*format,arg1,arg2,...***)**

Diese beiden Funktionen entsprechen den Funktionen **fprintf** und **fscanf**, mit der Ausnahme, daß kein Filepointer angegeben wird.

Beispiel:
```
main()
{
int n;
float s;
char name[40];
scanf("%s %d %f", name,&n,&s);
printf("\nName:%-20s Nr. %-4d
        Gehalt: %8.2f M", name,n,s);
printf("\n\nEnde
des Programms\n");
}
```

Mit **scanf** wird eine Eingabe angefordert. Es wird folgende Zeichenkette eingegeben:

Meier 713 850.6<CR>

Auf dem Terminal erscheinen folgende Ausgaben:

Name: Meier Nr.713 Gehalt: 850,60 M

Ende des Programms

10.5. Formatumwandlung im Speicher

Analog zu den Funktionen **fprintf** und **fscanf** existieren zwei weitere Funktionen, **sprintf** und **sscanf**, die die gleichen Formatumwandlungen vornehmen, aber die Zeichenketten nicht ein- und ausgeben, sondern die Zeichenketten im Speicher ablegen bzw. im Speicher voraussetzen. Diese Funktionen werden wie folgt aufgerufen:

sprintf(*string,format,arg1,arg2,...***)**
sscanf(*string,format,arg1,arg2,...***)**

wobei
string die Zeichenkette spezifiziert, die bei **sprintf** das Ergebnis der Formatwandlung aufnehmen soll bzw. bei **sscanf** die umzuwandelnde Zeichenkette enthält,
format die Formatbeschreibung (siehe 10.4.) und
argi die Argumente (siehe 10.4.) angeben.

10.6. Fehlerausgaben, Programmbeendigungen und Zusatzfunktionen

Für Fehlerausgaben und Programmbeendigungen wird meistens die Funktion

error("*string***")**

benutzt, wobei *string* die Fehlernachricht spezifiziert. Die Fehlernachricht wird auf **stderr** ausgegeben. Die Files werden ordnungsgemäß abgeschlossen und das Programm abgebrochen. (Für den Parameter **"***string***"** kann auch eine Parameterspezifikation wie in der **printf**-Funktion stehen.)
Ein Programm kann auch mit der Funktion **exit(***n***)** abgebrochen werden, wobei *n* einen Fehlerkode spezifiziert. Eine Null (0) zeigt eine fehlerfreie Ausführung an. Auch bei **exit** werden alle Files ordnungsgemäß abgeschlossen. Vor **exit** kann eine Aufbereitung einer Fehlermitteilung mit **fprintf** erfolgen.

Beispiel:
```
fprintf(stderr,"\nFile%s kann
nicht eröffnet werden
\n", name);
exit(1);
```

Bei der Eingabe von Zeichen kann mit der Funktion

ungetc(*c,fp***)**

das jeweils letzte Zeichen wieder in das File zurückgeschrieben und somit das File modifiziert werden. Hierbei haben die Parameter folgende Bedeutung:
c – zurückzuschreibende Variable
fp – Filepointer des zu modifizierenden Files.

10.7. Elementare E/A-Funktionen

Die elementaren E/A-Funktionen gestatten im Betriebssystem UNIX die unmittelbare Verarbeitung von beliebigen physischen Datenträgern im sequentiellen Zugriff und Direktzugriff. Der Trans-

fer wird dabei direkt zwischen dem E/A-Puffer und dem E/A-Gerät organisiert. Folgende Funktionen stehen zur Verfügung:

fd=**open(***name,rwmode***):**
Eröffnen eines Files
fd=**creat(***name,pmode***):**
Erstellen eines neuen Files
na=**read(***fd,buf,n***):**
Lesen eines Blockes
na=**write(***fd,buf,n***):**
Schreiben eines Blockes
close(*fd***):**
Schließen eines Files
unlink(*name***):**
Löschen eines Files
lseek(*fd,offset,position***):**
Positionieren auf ein Byte

wobei
fd – Filedeskriptornummer für das File,
name – Filespezifikation,
rwmode – Zugriffsart:
0 – nur Lesezugriff
1 – nur Schreibzugriff
2 – Lese- und Schreibzugriff
pmode – Fileschutzfestlegung (neun Bits):

rew	rew	rew
Eigentümer	Gruppenmitglieder	Übrige

r =**1** – Lesen erlaubt
e =**1** – Ausführen erlaubt
w=**1** – Schreiben erlaubt
buf – E/A-Puffer
n – Blockgröße (in der Regel 512 Byte)
na – aktuelle Blockgröße in Byte (tatsächlich ein- oder ausgegebene Byteanzahl)
offset – Nummer des Bytes, auf das positioniert werden soll. Diese Angabe muß im **long**-Format spezifiziert werden!
position – Fileposition als Basis für *offset:*
0 = Fileanfang
1 = aktuelle Fileposition
2 = Fileende

Der Parameter *fd* ist eine Filedeskriptornummer, die nicht mit dem Filepointer *fp* verwechselt werden darf. *fd* ist immer eine positive Integerzahl. Die Nummern **0**, **1** und **2** sind für die Files **stdin**, **stdout** und **stderr** vergeben. Werden weitere Files eröffnet, so werden weitere Nummern vergeben. Wenn ein File nicht eröffnet werden kann, so wird *fd* auf **–1** gesetzt.

Der Parameter *na* wird auch zur Fehlererkennung verwendet. Ein Wert **–1** zeigt einen Ein- bzw. Ausgabefehler und ein Wert **0** bei Eingabefiles das Fileende an.

Beispiel:

```
#include<stdio.h>
main()
{
char block[512];
int file1,file2;
if ((file1=open("TEST.TXT",0))<0)
error ("File 1 nicht vorhanden");
if ((file2=creat
("TEST.DOC",0766))<0)
   error ("File 2 kann nicht
   angelegt werden");
while ((n1=read(file1,block,
512))!=0)
{if (n1<0) error ("Eingabefehler");
if ((n2=write(file2,block,n1)!=n1)
error ("Ausgabefehler");}
close (file1); close (file2);
}
```

11. Methodik der Programmentwicklung

Unter fast allen Betriebssystemen erfolgt die Verarbeitung von Quellprogrammen, die in der Programmiersprache C geschrieben sind, nach dem Prinzip der zweistufigen Programmentwicklungsmethodik (Bild 1). Dieses Prinzip schließt die Verwendung eines Zwischenkodes, des sogenannten Objektkodes, ein. Jeder Quellmodul *fname*.**C** wird in der ersten Verarbeitungsstufe durch den C-Compiler **CC** in einen Objektmodul
fname.**OBJ** übersetzt. Unter einem Quellmodul wird allgemein eine Übersetzungseinheit des Compilers verstanden. Der C-Compiler läßt zu, daß ein Quellmodul eine oder mehrere komplette Funktionen enthält. Der Objektmodul enthält im wesentlichen das in den Maschinenkode des Zielrechners übersetzte C-Programm sowie Steuerinformationen über die Definition globaler Symbole und die Bezugnahme auf globale (externe) Symbole. Während der Übersetzung eines Quellmoduls kann der C-Compiler auf das File *STDIO.H* mit der Definition des Filedeskriptorblockes und der Filepointer für die Standardgeräte Bezug nehmen. Ebenso können beliebige andere Quellfiles mit der **#include**-Anweisung vom C-Compiler in den Quellmodul eingefügt werden.

In der zweiten Verarbeitungsstufe werden die getrennt übersetzten Objektmoduln durch den Programmverbinder (**LINK** – Linker oder **TKB** – Taskbuilder) zu einem ausführbaren Maschinenkodeprogramm (*fname*.**EXE** oder *fname*.**TSK**) verbunden. Dabei nimmt der Programmverbinder auch auf die Objektmodulbibliothek des C-Laufzeitsystems (**CSLIB.OLB** oder **C.OLB**) und gegebenenfalls auch auf private Objektmodulbibliotheken Bezug. Aus den Objektmodulbibliotheken werden nur die Objektmoduln verwendet, die benötigt werden, um noch offene Bezugnahmen auf externe (globale) Symbole auflösen zu können. An den bezugnehmenden Stellen in den verschiedenen Objektmoduln ergänzt der Programmverbinder die absoluten virtuellen oder physischen Adressen der externen (globalen) Symbole. Das Ergebnis der Programmverbindung ist der ausführbare Lademodul, der alle benötigten Hilfsfunktionen beinhaltet und keine offenen Bezugnahmen auf externe (globale) Symbole mehr enthält.

Die Aufbereitung der C-Quellmoduln erfolgt mit einem Standard-Editor des jeweiligen Betriebssystems. Da der C-Compiler keine Drucklisten des Quellmoduls erstellt, sondern nur eine Fehlerliste ausgibt, müssen für die Ausgabe der Druckliste des Quellmoduls die Betriebsfunktionen wie **TYPE**, **PRINT** oder **PIP** genutzt werden.

Der Programmverbinder erstellt bei Bedarf eine Druckliste, die die eingebundenen Objektmoduln und die Hauptspeicheraufteilung ausweist.

Der Start des Lademoduls erfolgt mit dem **RUN**-Kommando des Betriebssystems. Spezielle Testhilfen werden für den Test von C-Programmen nicht zur Verfügung gestellt. Zweckmäßigerweise bedient man sich für eine Testunterstützung der **printf**-Funktion, mit der man für Testzwecke die Zustände von internen Variablen an bestimmten Stellen des Algorithmus ausgibt. Später werden diese **printf**-Anweisungen entweder wieder aus dem Programm gelöscht oder in Kommentare umgewandelt.

Für die Verarbeitung eines C-Programms unter einem bestimmten Betriebssystem muß man sich in jedem Fall mit den entsprechenden Anweisungen zur Benutzung der verschiedenen Betriebssystemkomponenten, des C-Compilers und des Programmverbinders vertraut machen.

Bild 1 Methodik der Programmentwicklung

Programmierung in C

Teil VI

Dr. Thomas Horn
Informatikzentrum des Hochschulwesens
an der Technischen Universität
Dresden

Im abschließenden Teil dieser Artikelserie werden einfachere und ein komplexeres Programmbeispiel behandelt, an denen vor allem der strukturierte Programmentwurf in der Programmiersprache C verdeutlicht werden soll. Als komplexeres Beispiel wurde ein Druckprogramm gewählt, das nach vorgegebenen Parametern die Seiteneinteilung und die Seitennumerierung ausführt. Die Seitennumerierung kann dabei wahlweise in arabischen und römischen Zahlen erfolgen. Für eine getrennte Numerierung der Kapitel und Anlagen kann vor der Seitennummer ein beliebiger Text stehen, der durch einen Bindestrich getrennt ist, z. B. A-10. Tabulatoren (HT) werden beim Druck durch Leerzeichen ersetzt, wobei der Tabulatorgrundwert frei wählbar ist. Als letztes Beispiel wird ein Programm zur Verarbeitung von Gleitkommazahlen vorgestellt, das eine beliebige Anzahl von Zahlen nach aufsteigender Folge sortieren kann. Zur Vereinfachung des Tests werden die Zahlen im Dialog eingegeben und nach der Sortierung zur Kontrolle gedruckt.

12. Programmbeispiele

12.1. Kopieren eines Textfiles (1. Beispiel)

Es soll ein Programm zum Kopieren eines Textfiles geschrieben werden. Textfiles sind Files, die beliebigen Text im ASCII-Format enthalten. Ein solches Programm kann also auch benutzt werden, um C-Programme auf beliebige Datenträger zu kopieren, u. a. zum Erstellen einer Druckliste auf einem Drucker. Da in diesem Programm die einzelnen Datensätze unverändert kopiert werden sollen, sind hierfür zweckmäßigerweise die E/A-Funktionen **fgets** und **fputs** zu verwenden. Der prinzipielle Programmaufbau leitet sich aus den Prinzipien der Fileverarbeitung ab:

```
#include <stdio.h>
static char s1[]="Eingabefilespezifikation: ",
            s2[]="Ausgabefilespezifikation: ";
main()
{
char feld[8Ø], sin[25], sout[25];
FILE *fin, *fout;
do {   /* Eroeffnen des Eingabefiles */
   printf("%s", s1);
   if (gets(sin) == NULL) exit();
   fin = fopen(sin, "r");
   if (fin == NULL)
      printf("Fehler in der Filespezifikation\n");
   }
while (fin == NULL);
do {   /* Eroeffnen des Ausgabefiles */
   printf("%s", s2);
   if (gets(sout) == NULL) exit();
   fout = fopen(sout, "w");
   if (fout == NULL)
      printf("Fehler in der Filespezifikation\n");
   }
while (fout == NULL);
/* Kopieren des Files satzweise */
while(fgets(feld, 8Ø, fin)) fputs(feld, fout);
fclose(fout);  /* Schliessen des Ausgabefiles */
fclose(fin);   /* Schliessen des Eingabefiles */
printf("Kopieren beendet\n");
}
```

Bild 2 Programm **TEST1.C** *zum Kopieren eines Textfiles*

```
>run test1
Eingabefilespezifikation: test.c
Fehler in der Filespezifikation
Eingabefilespezifikation: test1.c
Ausgabefilespezifikation: lp:
Kopieren beendet

>
```

Bild 3 Dialog der Abarbeitung des Programms **TEST1.C**

– Eröffnen der verwendeten Files
– Kopieren des Files
– Schließen der verwendeten Files

und ist mit der Struktogrammtechnik als Folge von Strukturblöcken in Bild 1 grafisch verdeutlicht.

Damit das Programm zum Kopieren beliebiger Files benutzt werden kann, soll vor der Fileeröffnung die Filespezifikation des Eingabe- und des Ausgabefiles vom Standardgerät **stdin** angefordert werden. Dazu wird die **printf**-Funktion für die Ausgabe der Eingabeanforderung und die **gets**-Funktion zum Einlesen der Filespezifikation benutzt.

Bei einer fehlerhaften Fileeröffnung wird angenommen, daß die Filespezifikation falsch ist, und die Abfrage der Filespezifikation soll wiederholt werden. Wird auf eine Eingabeanforderung mit **EOF** (Tastenkombination **CTRL/Z**) geantwortet, so soll das Programm abgebrochen werden.

Bei erfolgreicher Beendigung des Kopierens soll die Ausschrift „Kopieren beendet" ausgegeben werden.

Der vollständige Quelltext des Programms **TEST1.C** ist in Bild 2 dargestellt. Bild 3 zeigt den Dialog während der Abarbeitung des Programms zum Kopieren des Programms **TEST1.C** auf den Drucker **LP:**. Bei der ersten Eingabe wurde aus Versehen ein nicht vorhandenes File **TEST.C** spezifiziert. (Der Bildschirmdialog wurde auf einem Drucker als Hardcopy-Gerät protokolliert.)

12.2. Kopieren eines Textfiles (2. Beispiel)

Im Quellprogramm des ersten Beispielprogramms ist zu ersehen, daß im Prinzip zweimal die gleiche Anweisungsfolge für die Fileeröffnungen verwendet wurde. Es erhebt sich sofort die Frage, ob man die Fileeröffnungen nicht in einer Funktion durchführen kann, die dann auch später noch in anderen Beispielen genutzt werden kann.

Zu diesem Zweck soll eine Funktion **vfopen** für die Eröffnung eines Files mit variablem Filenamen programmiert werden. Die Funktion benötigt offensichtlich 3 Parameter:

– Zeichenkette der Eingabeanforderung (*anf*)
– Adresse des Filepointers (*fp*)
– Zugriffsart für die Fileeröffnung (*mode*).

Das Programm der Funktion *vfopen* ist in Bild 4 dargestellt. Es gilt hier als besondere Schwierigkeit zu beachten, daß *fp* ein Zeiger

Eingabe der Filespezifikation und Eröffnen des Eingabefiles
Eingabe der Filespezifikation und Eröffnen des Ausgabefiles
while (kein EOF)
Kopieren eines Satzes
Schließen des Ausgabefiles
Schließen des Eingabefiles

Bild 1 Struktogramm des Programms zum Kopieren eines Textfiles

```
#include <stdio.h>
vfopen(anf,fp,mode)
char *anf, *mode;
FILE **fp;
{
char sf[25];
do {
   printf ("%s", anf);
   if (gets(sf) == NULL) exit();
   if ((*fp = fopen(sf, mode)) != NULL) return;
   printf("Fehler in der Filespezifikation\n");
   }
while(1);
}
```

Bild 4 Programm der Funktion **vfopen**

```
#include <stdio.h>
main()
{
char feld[8Ø];
FILE *fin, *fout;
/* Eroeffnen der Files */
vfopen("Eingabefilespezifikation: ",&fin,"r");
vfopen("Ausgabefilespezifikation: ",&fout,"w");
/* Kopieren des Files satzweise */
while(fgets(feld, 8Ø, fin)) fputs(feld, fout);
fclose(fout);   /* Schliessen des Ausgabefiles */
fclose(fin);    /* Schliessen des Eingabefiles */
printf("Kopieren beendet\n");
}
```

Bild 5 Programm **TEST2.C** *zum Kopieren eines Textfiles*

auf einen Filepointer ist. Deshalb ist er auch in Zeile 4 als Zeiger auf einen Zeigertyp definiert, und in Zeile 10 wird über den Zeiger *fp* auf den Filepointer indirekt zugegriffen. Durch Anwendung der Funktion **vfopen** ergibt sich eine wesentliche Vereinfachung des Programms **TEST1.C**, das als Programm TEST 2.C in Bild 5 dargestellt ist. Dieses Beispiel macht deutlich, wie sich der modulare Aufbau von größeren Programmsystemen auf ihre Transparenz, Einfachheit und Fehlerfreiheit auswirkt.

12.3. Kopieren eines Textfiles (3. Beispiel)

In diesem Beispiel soll die gleiche Aufgabenstellung noch einmal gelöst werden. Im Unterschied zu den ersten beiden Beispielen soll jetzt aber der Dialog zur Abfrage der Filespezifikationen durch die Übernahme der Filespezifikationen aus der Kommandozeile ersetzt werden.
Das Programm soll wie folgt gestartet werden:

>**TEST3** *infile outfile*

Wenn die Ausgabefilespezifikation *outfile* nicht angegeben wird, soll die Ausgabe auf das Standardausgabegerät **stdout** erfolgen. Wird kein Parameter spezifiziert, so soll das Programm ohne ein Resultat beendet werden. Für das Beispiel aus 12.1. zum Kopieren des Files **TEST1.C** auf den Drucker **LP:** müßte dann die Kommandozeile wie folgt spezifiziert werden:

>**TEST3 TEST1.C LP:**

Der vollständige Programmtext ist in Bild 6 dargestellt und zeigt, wie die Parameter aus der Kommandozeile zu verwenden sind.

12.4. Drucken eines Textfiles mit Seitennumerierung

Aufbauend auf den ersten einfachen Beispielen soll nun das Kopieren des Textfiles um zwei Funktionen erweitert werden:

- Im Text enthaltene Tabulatoren sollen durch Leerzeichen ersetzt werden. Diese Funktion ist nützlich, da erstens viele Drucker das Steuerzeichen „Tabulatorsprung" (*HT* – **011**) nicht interpretieren können und zweitens man die Möglichkeit zum Steuern des Tabulatorsprungs hat, um wieviele Druckpositionen maximal gesprungen werden soll.

```
#include <stdio.h>
main(argc, argv)
int argc; char *argv[];
{
char feld[80];
FILE *fin, *fout;
/* Eroeffnen der Files */
if (argc == 1) exit();
if ((fin = fopen(argv[1], "r")) == NULL)
   error ("Eingabefile kann nicht eroeffnet werden\n");
if (argc == 2) fout = stdout;
else if ((fout = fopen(argv[2], "w")) == NULL)
   error ("Ausgabefile kann nicht eroeffnet werden\n");
/* Kopieren des Files satzweise */
while(fgets(feld, 80, fin)) fputs(feld, fout);
fclose(fout);  /* Schliessen des Ausgabefiles */
fclose(fin);   /* Schliessen des Eingabefiles */
printf("Kopieren beendet\n");
}
```

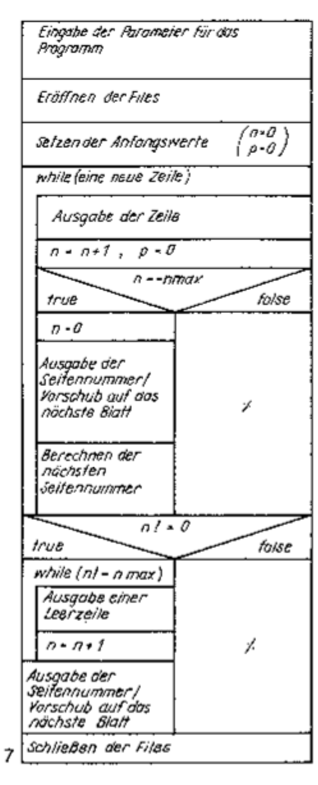
Bild 6 Programm **TEST3.C** *zum Kopieren eines Textfiles*

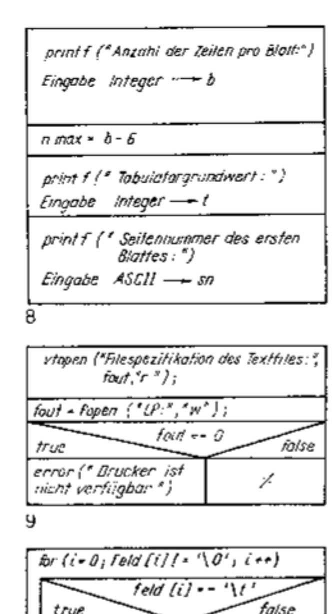
Bild 7 Struktogramm des Programms zum Drucken von Textfiles mit Seitennumerierung

Bild 8 Struktogramm für die Eingabe der Programmparameter

Bild 9 Struktogramm für das Eröffnen der Files

Bild 10 Struktogramm für die Ausgabe einer Zeile





Bei den Rechenanlagen des SKR ist dieser Tabulatorgrundwert konstant 8, was für eine Reihe von Anwendungen nicht immer zweckmäßig ist.

- Ausgehend vom eingestellten Zeilenabstand (4 Zeilen/Zoll, 6 Zeilen/Zoll usw.) und von der Papiergröße soll beim Ausgeben des Textfiles eine Seiteneinteilung erfolgen und auf jeder Seite unten in der Mitte eine Seitennummern gedruckt werden.

Da größere Texte zweckmäßigerweise in mehrere Files unterteilt werden, muß die Seitennummer der ersten Seite frei wählbar sein. Die Seitennummer der Folgeseiten sollen automatisch gebildet werden.
Gleichzeitig sollen neben arabischen auch römische Seitennummern sowie eine getrennte Numerierung nach Abschnitten, Kapiteln bzw. Anlagen zugelassen werden, indem vor der variablen Seitennummer durch Bindestrich getrennt ein konstanter Text stehen darf.
In der ersten Entwurfsetappe werden die Funktionen des Programms im groben spezifiziert, wie Eingabe der Parameter des Programms, Eröffnen der Files, Setzen der Anfangswerte für den Zeilenzähler **n** und den Druckpositionszähler **p**, Kopieren des Textfiles unter Beachtung der zwei geforderten Funktionen, Auffüllen der letzten Seite mit Leerzeilen und Schließen der Files.
Charakteristisch für das Programm sind vor allem der vierte und fünfte Strukturblock. Der vierte Strukturblock besteht aus einer Abweisschleife, die solange ausgeführt wird, wie noch eine Textzeile verfügbar ist. Unter Beachtung der Substition der Tabulatoren ist die Textzeile auszugeben. Nach der Ausgabe ist der Zeilenzähler **n** zu inkrementieren und der Positionszähler **p zu löschen. Wenn die maximale Zeilenzahl erreicht wurde, muß** der Zeilenzähler **n** gelöscht werden, die Seitennummer gedruckt werden, ein Vorschub auf das nächste Blatt erfolgen und die nächste Seitennummer berechnet werden.
Im fünften Strukturblock wird der Zeilenzähler **n** getestet. Bei **n** ungleich **0** ist die letzte Seite unvollständig. Solange, bis **n** gleich **nmax** ist, werden Leerzeilen ausgegeben. Anschließend wird die Seitennummer gedruckt und ein Vorschub auf das nächste Blatt realisiert. Das Strukturprogramm für diesen grundsätzlichen Programmablauf ist in Bild 7 dargestellt.
Im weiteren werden wir nun dieses Struktogramm schrittweise verfeinern. Die Eingabe der Programmparameter (Bild 8) wird durch die einzugebenden Parameter konkretisiert:

- Anzahl der Zeilen pro Blatt **b**. Für die Seitennummer, den unteren und den oberen Blattrand sollen **6** Zeilen verwendet werden, so daß sich **nmax** aus **b—6** ergibt.
- Tabulatorgrundwert **t**.
- Seitennummer des ersten Blattes **sn**.

Das Eröffnen der Files (Bild 9) wird durch die zu eröffnenden Files konkretisiert. Für das Eröffnen des Textfiles wird die Funktion **fopen()** verwendet. Die Druckereröffnung erfolgt ganz normal über die Funktion **vfopen()**.
Die Ausgabe der Zeile (Bild 10) erfolgt zeichenweise. Jedes Zeichen wird auf *HT*(**\t**) analysiert. Bei einem *HT* wird ein Leerzeichen ausgegeben. Ist die neue Position nicht durch den Tabulatorgrundwert teilbar, so werden weitere Leerzeichen ausgegeben.
Analog können alle anderen Strukturblöcke konkretisiert und formalisiert werden. Im Ergebnis dieser zweiten Programmentwurfsetappe entsteht dann das in Bild 11 dargestellte verfeinerte Struktogramm. Die Ausgabe der Seitennummer und des Blattwechsels kann mit einer **fprintf**-Anweisung erfolgen, wenn eine entsprechende Textkonstante **tc** mit den erforderlichen Steuerzeichen benutzt wird. Die Berechnung der

neuen Seitennummer ist ein recht komplexes und im Prinzip auch selbständiges Problem, so daß festgelegt wurde, eine Funktion **seite()** dafür zu definieren.
Das dem Bild 11 äquivalente C-Programm ist in Bild 12 dargestellt.
Die Funktion **seite()** dient der Berechnung einer neuen Seitennummer (Bild 13). Bei einer leeren Zeichenkette ist keine Erhöhung der Seitennummer erforderlich, und es erfolgt sofort ein Rücksprung. Nach diesem Test muß ein Bindestrich gesucht werden. Wurde ein Bindestrich gefunden, so wird der Index **i** auf das nächste Zeichen nach dem Bindestrich gesetzt. Anderenfalls wird der Index **i** auf das nächste Zeichen nach dem Bindestrich gesetzt. Ist das erste Zeichen, worauf der Index **i** zeigt, eine Ziffer von **'0'–'9'**, so liegt eine arabische Zahl vor, anderenfalls eine römische Zahl. Die arabische Zahl wird gleich in der ASCII-Zeichenkette erhöht. Es wird die letzte Ziffer gesucht und inkrementiert. Wenn der Kode größer wird als der Kode der Ziffer **'9'**, so wird die Ziffer auf **'0'** gesetzt und die vorhergehende Ziffer inkrementiert usw. Wenn die Zahl um eine Ziffer größer wird, so muß sie um ein Zeichen nach rechts verschoben werden, und an vorderster Stelle muß eine **'1'** eingeblendet werden.

```
printf ("Anzahl der Zeilen pro Blatt : ")
Eingabe  Integer ——► b
nmax = b - 6
printf (" Tabulatorgrundwert : ")
Eingabe  Integer ——► t
printf (" Seitennummer des ersten Blattes : ")
Eingabe  ASCII ——► sn
vfopen ("Filespezifikation des Textfiles",fout,"w"
fout = fopen (" LP: ", "w")
fout == 0 : true / false
  true:  error (" Drucker ist nicht verfügbar")
  false: ./.
n = 0 , P = 0
while (fgets (feld , 100 , fin))
  for (i = 0; feld [i] != '\0'; i++
    feld [i] == 't' : true / false
      true:  putc ('b', fout)
             p = p + 1
             while (p % t != 0
               putc ('b', fout)
               p = p + 1
      false: putc (feld[i], fout)
             p = p + 1
  n = n + 1 , p = 0
  n == nmax : true / false
    true:  n = 0
           fprintf (fout, tc, sn)
           seite (sn)
    false: ./.
n != 0 : true / false
  true:  while (n != nmax)
           putc ('\n', fout)
           n = n + 1
         fprintf (fout, tc, sn)
  false: ./.
fclose (fin), fclose (fout)
```

Bild 11 Verfeinertes Struktogramm des Programms zum Drucken von Textfiles

Das Inkrementieren einer römischen Zahl ist wesentlich komplizierter. Deshalb wurde hier der Weg der Konvertierung einer römischen Zahl in einen Binärwert (Integer), des Inkrementierens des Binärwertes und der Konvertierung des Binärwertes in die römische Zahlendarstellung gewählt. Die Konvertierung aus der römischen bzw. in die römische Zahlendarstellung sind universell anwendbare Algorithmen. Darum wurden diese Algorithmen als gesonderte Funktionen implementiert:

crtb() – Konvertierung aus dem Römischen ins Binäre
cbtr() – Konvertierung aus dem Binären ins Römische

Die Funktionen **seite()**, **crtb()** und **cbtr()** sind in den Bildern 14, 15 und 16 dargestellt.
Bild 17 zeigt den Dialog der Abarbeitung des Programms **druck**. Als Beispiel wurde ein Programm **sort.c** ausgedruckt. Das Bild 18 zeigt die Ausgabe des Programms **sort.c** mit dem Tabulatorgrundwert **2** und Bild 19 mit dem Tabulatorgrundwert **5**. Die Anweisungen in der **for**-Schleife wurden jeweils mit 1, 2 oder 3 Tabulatorsprüngen (*HT*) eingerückt. Die Seitennumerierung ist in Bild 12 gezeigt.

```
static char tc[]="\n\n          \
                %s\n\n\n\n";
#include <stdio.h>
main()
/* Programm zum Drucken von Textfiles
mit Seitennumerierung */
{
/* Vereinbarung der Pointer, Variablen, Felder */
FILE *fin,*fout;
int n,nmax,b,t,p,i;
char sn[1Ø],feld[1ØØ],conv[2Ø];
/* Eingabe der Programmparameter */
printf("Anzahl der Zeilen pro Blatt: ");
gets(conv);
sscanf(conv,"%d",&b);
nmax=b-6; /* druckbare Zeilen pro Blatt */
printf("Tabulatorgrundwert: ");
gets(conv);
sscanf(conv,"%d",&t);
printf("Seitennummer des ersten Blattes: ");
gets(sn);
/* Eroeffnen des Textfiles */
vfopen("Filespezifikation des Textfiles: ",&fin,"r");
/* Eroeffnen des Druckers */
if((fout=fopen("LP:","w"))==NULL)
    error("Drucker ist nicht verfuegbar");
/* Setzen der Anfangswerte */
n=Ø;  /* Zeilenzaehler */
p=Ø;  /* Positionszaehler */
/* Ausgabe des Textes mit Aufbereitung */
while(fgets(feld,1ØØ,fin))  /* Zeile einlesen */
{
    /* Ausgabe einer Zeile */
    for(i=Ø;feld[i]!='\Ø';i++)
    /* Analyse auf Tabulator */

                    6–1

        if(feld[i]=='\t')
        {
            /* Ausgabe eines Leerzeichens und
            positionieren auf die naechste durch
            t teilbare Position */
            putc(' ',fout); p++;
            while(p%t!=Ø) { putc(' ',fout);p++; }
        }
        else
        {
            /* Ausgabe aller anderen Zeichen */
            putc(feld[i],fout);p++;
        }
    n++;       /* Zeilenzaehler +1 */
    p=Ø;       /* Positionszaehler =Ø */
    /* Analyse auf Blattende */
    if(n==nmax)
    {
        n=Ø;  /*Zeilenzaehler loeschen */
        fprintf(fout,tc,sn); /*Drucken Seitennummer*/
        seite(sn);  /* Erhoehen der Seitennummer */
    }
}
/* Auffuellen der letzten Seite mit Leerzeilen */
if(n!=Ø)
{
    while(n!=nmax)
        {  putc('\n',fout);n++; }
    /* Drucken der letzten Seitennummer */
    fprintf(fout,tc,sn);
}
/* Schliessen der Files */
fclose(fin); fclose(fout);
}

                    6–2
```

Bild 12 Text des Programms zum Drucken von Textfiles

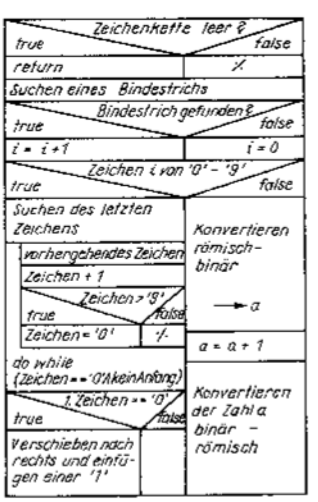


Bild 13 Struktogramm des Moduls für die Erhöhung der Seitennummern

12.5. Sortieren von Zahlen

Zum Abschluß sei ein kleines Programm zur Verarbeitung von **float**-Zahlen am Beispiel eines häufig angewendeten Algorithmus zum

```
seite(p)
/* Berechnung der neuen Seitennummer */
/* p -- Textpuffer mit Seitennummer */
char p[];
{
int i, j, a;
if (p[Ø]=='\Ø') return; /*keine Seitennummer*/
for (i = Ø; p[i] != '\Ø' && p[i] != '-'; i++);
/* wenn Seitennummer mit "-", dann Index i
auf naechstes Zeichen nach "-" */
if (p[i] == '-') i++; else i = Ø;
/* Test auf arabisch, ansonsten roemisch */
if ((p[i] >= 'Ø') && (p[i] <= '9'))
{ /* arabisch */
    /* Suchen des Zeichenkettenendes */
    for (j=i; p[j] != '\Ø'; j++);
    /* Erhoehen der letzten Ziffer, wenn =Ø,
    dann erhoehen der vorletzten Ziffer usw.*/
    do
    {
        j--; p[j]++;
        if(p[j]> '9') p[j]= 'Ø';
    }
    while (( p[j] == 'Ø') && ( j > i));
    /* wenn erste Ziffer =Ø, dann verschieben
    nach rechts und einschieben einer 1 */
    if ((j == i) && (p[j] == 'Ø'))
    {
        for (j = i; p[j] != '\Ø'; j++);
        for (;j >= i; j--) p[j+1]= p[j];
        p[i] = '1';
    }
}
else
{ /* roemisch */
    a=crtb(p,i);
    a++;
    cbtr(a,p,i);
}
return;
}
```

Bild 14 Text des Moduls für die Erhöhung der Seitennummern

```
crtb(p,i)
/* Konvertierung Roemisch in Binaer */
/* p -- Textpuffer mit roemischer Zahl
   i -- Anfangsposition im Textpuffer
   Funktionswert -- aequivalenter Binaerwert */
int i; char p[];
{
int a;
a=Ø;
/* Tausender */
while (p[i]=='M') { i++; a+=1ØØØ; }
/* Hunderter */
if ((p[i]=='C') && (p[i+1]=='M')) { i+=2; a+=9ØØ; }
if ((p[i]=='C') && (p[i+1]=='D')) { i+=2; a+=4ØØ; }
if (p[i]=='D') { i++; a+=5ØØ; }
while (p[i]=='C') { i++; a+=1ØØ; }
/* Zehner */
if ((p[i]=='X') && (p[i+1]=='C')) { i+=2; a+=9Ø; }
if ((p[i]=='X') && (p[i+1]=='L')) { i+=2; a+=4Ø; }
if (p[i]=='L') { i++; a+=5Ø; }
while (p[i]=='X') {i++; a+=1Ø; }
/* Einer */
if ((p[i]=='I') && (p[i+1]=='X')) { i+=2; a+=9; }
if ((p[i]=='I') && (p[i+1]=='V')) { i+=2; a+=4; }
if (p[i]=='V') { i++; a+=5; }
while (p[i]=='I') { i++; a+=1; }
return (a);
}
```

Bild 15 Text des Programms für die Konvertierung Römisch–Binär

```
cbtr(a,p,i)
/* Konvertierung Binaer in Roemisch */
/* a -- zu konvertierender Wert
   p -- Textpuffer fuer die roemische Zahl
   i -- Anfangsposition im Textpuffer */
int a, i; char p[];
{
/* Tausender */
while (a>=1000) { a-=1000; p[i++]='M'; }
/* Hunderter */
if (a>=900) { a-=900; p[i++]='C'; p[i++]='M'; }
if (a>=500) { a-=500; p[i++]='D'; }
if (a>=400) { a-=400; p[i++]='C'; p[i++]='D'; }
while (a>=100) { a-=100; p[i++]='C'; }
/* Zehner */
if ( a>=90) { a-=90; p[i++]='X'; p[i++]='C'; }
if (a>=50) { a-=50; p[i++]='L'; }
if (a>=40) { a-=40; p[i++]='X'; p[i++]='L'; }
while (a>=10) { a-=10; p[i++]='X'; }
/* Einer */
if (a>=9) { a-=9; p[i++]='I'; p[i++]='X'; }
if (a>=5) { a-=5; p[i++]='V'; }
if (a>=4) { a-=4; p[i++]='I'; p[i++]='V'; }
while (a>=1) { a-=1; p[i++]='I'; }
p[i++]='\0';  /* Zeichenkettenende */
return;
}
```

Bild 16 Text des Programms für die Konvertierung Binär–Römisch

```
>run druck
Anzahl der Zeilen pro Blatt: 68
Tabulatorgrundwert: 5
Seitennummer des ersten Blattes: 1
Filespezifikation des Textfiles: sort.c
>
```

Bild 17 Eingabedialog des Programms zum Drukken von Textfiles

```
main()
/* Sortieren von max 100 Zahlen */
{
int i,j,k;
static float a[100], z;
printf ("Anzahl der Zahlen: ");
scanf ("%d", &k);
printf ("Geben sie %d Zahlen getrennt durch\
 SP, HT und/oder CR ein:\n", k);
for (i=0; i<k; i++) scanf("%f", &a[i]);
for (i=0; i<k-1; i++)
  for (j=i+1; j<k; j++)
    if (a[i]>a[j])
      {
      z=a[i];
      a[i]=a[j];
      a[j]=z;
      }
printf ("Ergebnisse:\n");
for (i=0; i<k; i++) printf ("  %f\n",a[i]);
}
```

Bild 18 Programm sort.c mit Tabulatorgrundwert 2 gedruckt

Sortieren von Zahlen gezeigt. Zum Testen des Algorithmus können die Tabellengröße und die Testzahlen mit **scanf**-Anweisungen eingelesen werden. Nach der Sortierung wird die Tabelle mit **printf**-Anweisungen ausgegeben.

Der Text des Sortierprogramms ist in Bild 18 und 19 abgedruckt. Von der Vergleichsoperation in Zeile 13 ist abhängig, ob die Zahlen in steigender oder fallender Folge sortiert werden sollen („größer als" – steigende Folge).

13. Schlußbemerkungen

In der vorliegenden Artikelreihe „Programmierung in C" wurde der Versuch unternommen, dem Leser in einer methodisch-didaktisch aufbereiteten Form die international weit verbreitete Programmiersprache C vorzustellen. Die Artikelreihe kann dem Leser deshalb als Selbststudienmaterial dienen, wobei gleichzeitig auch auf Vollständigkeit Wert gelegt wurde, so daß es auch für die praktische Arbeit genutzt werden kann. Die abschließenden Beispiele verfolgten das Ziel, dem Leser bei der Programmierung eigener Aufgaben eine methodische Unterstützung zu geben.

Abschließend möchte ich hoffen, mit dieser Artikelreihe den Interessen vieler Leser entgegengekommen zu sein, eine Anregung zur intensiveren Beschäftigung mit der Programmiersprache C gegeben und somit auch einen Beitrag zu ihrer weiteren Verbreitung in unserer Republik geleistet zu haben.

✉ KONTAKT ☎

Informatikzentrum des Hochschulwesens an der Technischen Universität Dresden, Mommsenstr. 13, Dresden, 8027; Tel. 4575303.

Die Programmiersprache C

Folienreihen HFR 913/914 als Lehrmittel für die Aus- und Weiterbilung

Vom Institut für Film, Bild und Ton werden ab 1988 zwei Folienreihen für die Aus- und Weiterbildung zur „Programmierung in C" mit einem Gesamtumfang von 30 Polyluxfolien angeboten.

Die Folienreihen sind so gestaltet, daß sie sowohl im Hochschulstudium als auch zur Weiterbildung in Betriebsakademien und KDT-Lehrgängen für Vorlesungen und Übungen einsetzbar sind. Insbesondere dienen sie zur methodischen Stützung einer kompletten Vorlesungsreihe zur Programmierung in C.

Spezifika einzelner C-Compiler unter bestimmten Betriebssystemen spiegeln sich in den Folienreihen nicht wider.

Die Folienreihe HFR 913 behandelt die Grundsprachelemente, die einfachen Datentypen, Operatoren, Ausdrücke und Steueranweisungen.

Die Folienreihe HFR 914 erläutert die höheren Datentypen wie Zeiger, Felder, Strukturen und Vereinigungen sowie die Funktionen im allgemeinen, die Ein- und Ausgabefunktionen im speziellen und den Makropräprozessor.

Die Folienreihen können ab sofort beim

VEB DLK Erfurt
Audiovisuelle Lehrmittel
Junker-Jörg-Str. 31
Berlin
1167

bestellt werden.

```
main()
/* Sortieren von max 100 Zahlen */
{
int i,j,k;
static float a[100], z;
printf ("Anzahl der Zahlen: ");
scanf ("%d", &k);
printf ("Geben sie %d Zahlen getrennt durch\
 SP, HT und/oder CR ein:\n", k);
for (i=0; i<k; i++) scanf("%f", &a[i]);
for (i=0; i<k-1; i++)
     for (j=i+1; j<k; j++)
          if (a[i]>a[j])
               {
               z=a[i];
               a[i]=a[j];
               a[j]=z;
               }
printf ("Ergebnisse:\n");
for (i=0; i<k; i++) printf ("  %f\n",a[i]);
}
```

Bild 19 Programm sort.c mit Tabulatorgrundwert 5 gedruckt

C-Programmierwettbewerb in MP

Ihr erworbenes Wissen zur Programmiersprache C können Sie in unserem Programmierwettbewerb, zu dem wir in MP 7/87, 2. Umschlagseite aufrufen, unter Beweis stellen. Zwei Aufgaben, die zu lösen sind, hat unser Autor für Sie ausgesucht. Unter den richtigen Einsendungen verlosen wir 10 Bücher des Titels „UNIX und C", der in diesem Jahr in unserem Verlag erschienen ist. *MP*

Der Autor unserer sechsteiligen Beitragsfolge zu C, Doz. Dr. sc. techn. Thomas Horn (39), studierte von 1966–1972 an der Elektrotechnischen Hochschule „W. I. Uljanow (Lenin)" in Leningrad in der Fachrichtung Elektronische Rechenanlagen. Von 1972 bis 1975 promovierte er an der gleichen Hochschule auf dem Gebiet der Betriebssysteme zum Dr.-Ing.

Seit 1975 ist er an der Ingenieurhochschule Dresden auf dem Gebiet der Software für Klein- und Mikrorechner tätig. 1983 promovierte er an der Technischen Universität Dresden auf dem Gebiet der Architektur von Betriebssystemen für Klein- und Mikrorechner zum Dr. sc. techn. 1985 wurde er an der Ingenieurhochschule Dresden zum Hochschuldozenten für die Programmierung von Mikrorechnern berufen. Seit der Gründung des Informatikzentrums des Hochschulwesens ist Doz. Dr. Horn an der Technischen Universität Dresden beschäftigt.

Sein Arbeitsgebiet liegt gegenwärtig vorrangig auf dem Gebiet der Gestaltung und Implementierung von Systemsoftware für moderne Kleinrechner. Dabei kommt der Anwendung der Systemprogrammiersprache C eine besondere Bedeutung zu.

Doz. Dr. Horn ist seit 1981 Leiter der Arbeitsgruppe Software und seit 1983 Vorsitzender des Problemrates der Kooperationsgemeinschaft SM/SM4.

Was erscheint demnächst in MP-Kurs?

In Heft 7/87 beginnen wir mit einer Beitragsreihe über *Programmierung in MACRO-SM*. Bereits in MP 8/87 wird ebenfalls mit einer neuen Folge, nämlich über REDABAS, begonnen. Eine Artikelserie wird also in der Regel nicht fortlaufend gedruckt, sondern alternierend mit anderen. Wir hoffen, damit mehr den Interessen aller Leser zu entsprechen. So muß ein Leser u. U. nicht mehr Monate „warten", bis ein für ihn interessantes Thema unter dieser Rubrik erscheint. Weiterhin werden u. a. Artikelfolgen vorbereitet zum Mikroprozessor(-system) K 1810 WM 86 (8086) und zu Turbo Pascal. *MP*